VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Systém pro měření pokrytí kód testy

System for Mesuring Code Coverage

2017 Bc. Jan Michálek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jan Michálek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Systém pro měření pokrytí kód testy
System for Mesuring Code Coverage

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je vytvořit zásuvný modul pro prostředí Eclipse pro měření pokrytí kódu prováděnými testy. Práce se zaměří se na netradiční pokrytí kódu jako jsou (pokrytí cest, pokrytí výrazů, LCSAJ, ...). Součástí práce bude průzkum problematiky měření pokrytí kódu v jazyce Java. Vytvářený zásuvný modul bude zaměřen na záznam pokrytí průběhu kódu a jeho vyhodnocení.

Zásuvný modul umožní:
1. Záznam průběhu kódu v podobě umožňující detailní vyhodnocení průběhu.
2. Prototypové vyhodnocení pokrytí pro zvolené techniky.
3. Prezentaci výsledků ve zvolené formě (vizualizaci ve zdrojových kódech nebo export).

Práce bude obsahovat:
1. Přehled dostupných řešení pro měření pokrytí kódu testy.
2. Přehled problematiky měření pokrytí a manipulaci s Java bytecode.
3. Analýzu a implementaci výše popsaného zásuvného modulu.
4. Programátorskou dokumentaci řešení s využitím diagramů jazyka UML.

Seznam doporučené odborné literatury:

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four): Návrh programů pomocí vzorů. Grada. Praha 2003. ISBN 8024703025
[2] Black, R. & Mitchell, J. L.. Advanced Software Testing - Vol. 3: Guide to the ISTQB Advanced Certification As an Advanced Technical Test Analyst. Rocky Nook, 2011. ISBN 1933952393, 9781933952390 Rocky Nook, 2011

Dále podle pokynů vedoucího práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 20. dubna 2017 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 20. dubna 2017 .....

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla. Zvláště mému vedoucímu Ing. David Ježek, Ph.D., který při nejasnostech vždy poradil a našel si na mě čas.

Abstrakt

Diplomová práce se zabývá návrhem a implementace prototypu modulu do vývojového prostředí Eclipse. Modul umožňuje záznam průběhu kódu pro pokrytí cest. Pokrytí následně vizualizuje. A poslední částí diplomové práce je přehled dostupných řešení. Modul využívá dostupnou knihovnu pro měření pokrytí Jacoco. Pro testování kódů se využívá framework JUnit.

Klíčová slova: Jacoco, pokrytí cest, JUnit, ASM, JVM, Java, bytecode

Abstract

This diploma thesis deals with the design and implementation of the prototype of the module into Eclipse development environment. The module allows you to record the progress of the code to path coverage. The coverage is then visualized. The next part is a simple presentation of the results. And the last part of the diploma thesis is an overview of available solutions. The module uses the available Jacoco Coverage Library. Code JUnit is used to test the codes.

Key Words: Jacoco, path coverage, JUnit, ASM, JVM, Java, bytecode

Obsah

Seznam použitých zkratek a symbolů

JDT	–	Java development Tools
AST	–	Abstract Syntax Tree
IDE	–	Integrated Development Environment
SW	–	Software
LCSAJ	–	Linear Code Sequence And Jump
IP	–	Internet Protocol
TCP	–	Transmission Control Protocol
JAR	–	Java ARchive
JDT	–	Java Development Tools
PDE	–	Plug-in Development Environment
DOM	–	Document Object Model

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Po průzkumu současných modulů pro pokrytí kódu nebo cest bylo zjištěno, že všechny moduly pro testování kódu pracují stejně a umožňují jenom pokrytí kódu. Proto bylo potřeba vytvořit modul, který dokáže detailně zaznamenat pokrytí cest a následně vizualizovat výsledky. Pro usnadnění práce existuje knihovna Jacoco, která dokáže zaznamenat pokrytí kódů. Ta funguje na principu vkládání vlastního Java bytecodu. Jelikož ale umožňuje jenom pokrytí kódu, musí se tato knihovna upravit tak, aby dokázala vyhodnotit pokrytí cest.

Diplomová práce se rozděluje na čtyři kapitoly.

První kapitola se zabývá průzkumem současných řešení pro známé programovací jazyky. Zde patří i co jednotlivá řešení umí.

Druhá kapitola se zabývá teorií testování. Co je to vůbec testování, jaké typy pokrytí existují a základní informace o frameworku JUnit.

Třetí kapitola se zabývá Java technologií. Je zde popsán virtuální stroj, který spouští Java programy. Dále je zde popsán Java bytecode. Jeho struktura a princip fungování.

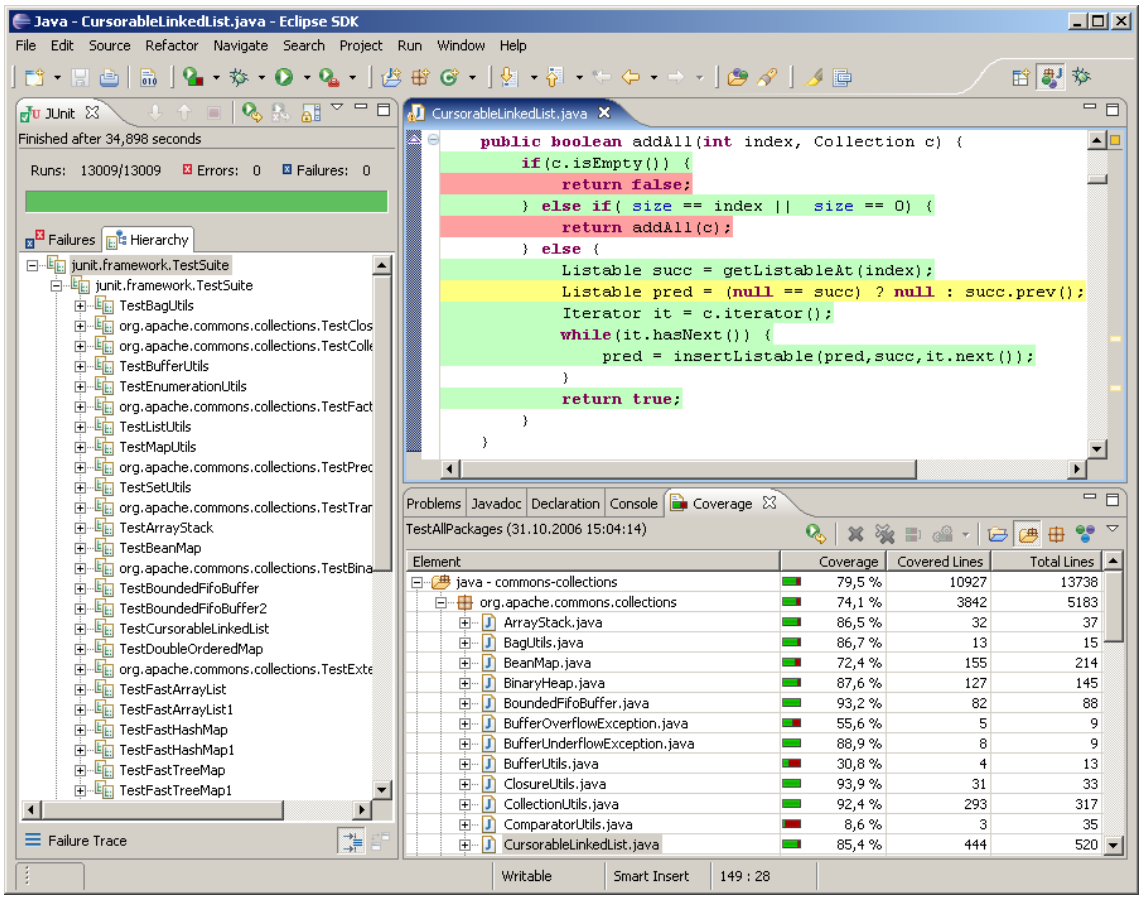
Čtvrtá kapitola se zabývá samotnou implementací a návrhem modulu pro pokrytí cest. Ta se dá rozdělit do několika podkapitol. Prvně se popisuje princip fungování Jacoco. Dále se popisuje úprava knihovny Jacoco. Další důležitou podkapitolou je vytvoření stromové struktury kódu a dále vyhodnocení pokrytí cest. A poslední podkapitola se zabývá modulem pro Eclipse.

2 Přehled dostupných řešení

Na internetu se nachází mnoho dostupných řešení pro měření pokrytí kódu. Každé řešení se, ale vztahuje k určitému programovacímu jazyku a pro určité IDE. Většina z nich, ale dokáže jenom určit jestli daný kód se prošel a případně kolikrát. A u větvení kódu vypsat jestli do bloku vešel a nebo ne. Moduly pro Eclipse se moc neliší a poskytují stejnou funkcionalitu.

2.1 EclEmma

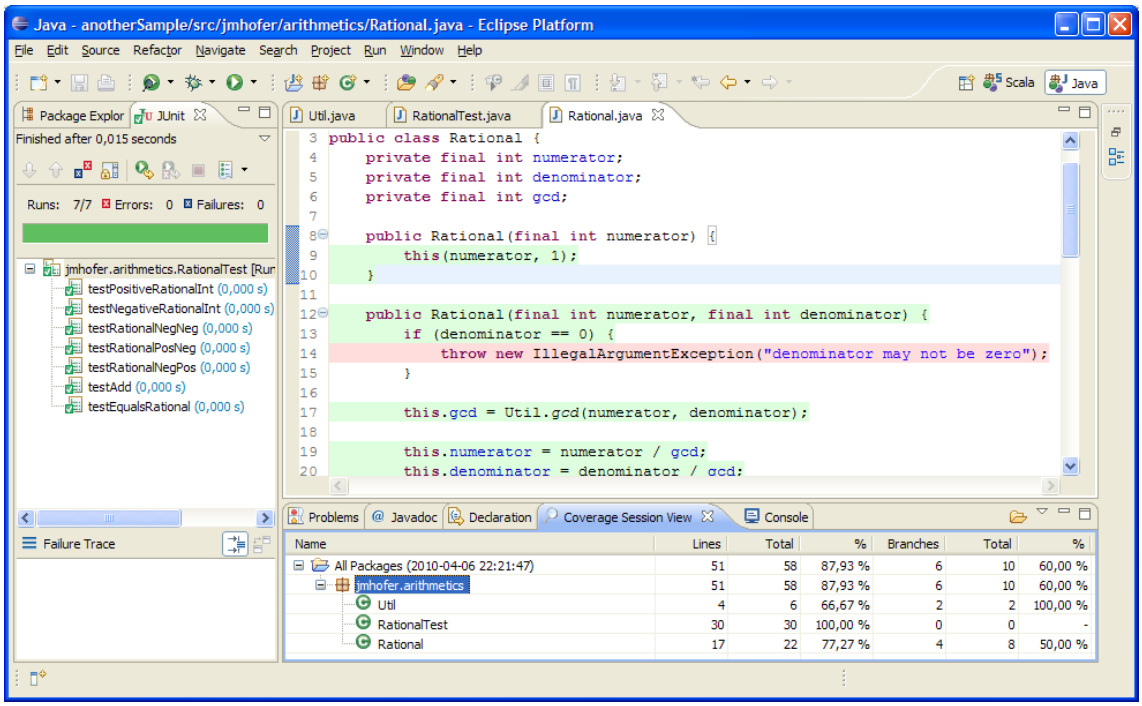
Neznámější a nejstahovanější modul do Eclipse pro Javu je „EclEmma“[1]. EclEmma umožňuje zaznamenávat pokrytí kódu jak obyčejným spuštění aplikace tak i pomocí testovacími framework. Jako je například „JUnit“. Umožňuje přímo do kódu vykreslit jestli se daný kód vykonal a u větvení vypsat jestli do bloku vešel a nebo ne. Dále vykreslí stromovou strukturu projektu u kterých vykreslí na kolik procent se vykonal. EclEmma podporuje nejnovější verzi Eclipsu.



Obrázek 1: Modul EclEmma pro Eclipse

2.2 eCobertura

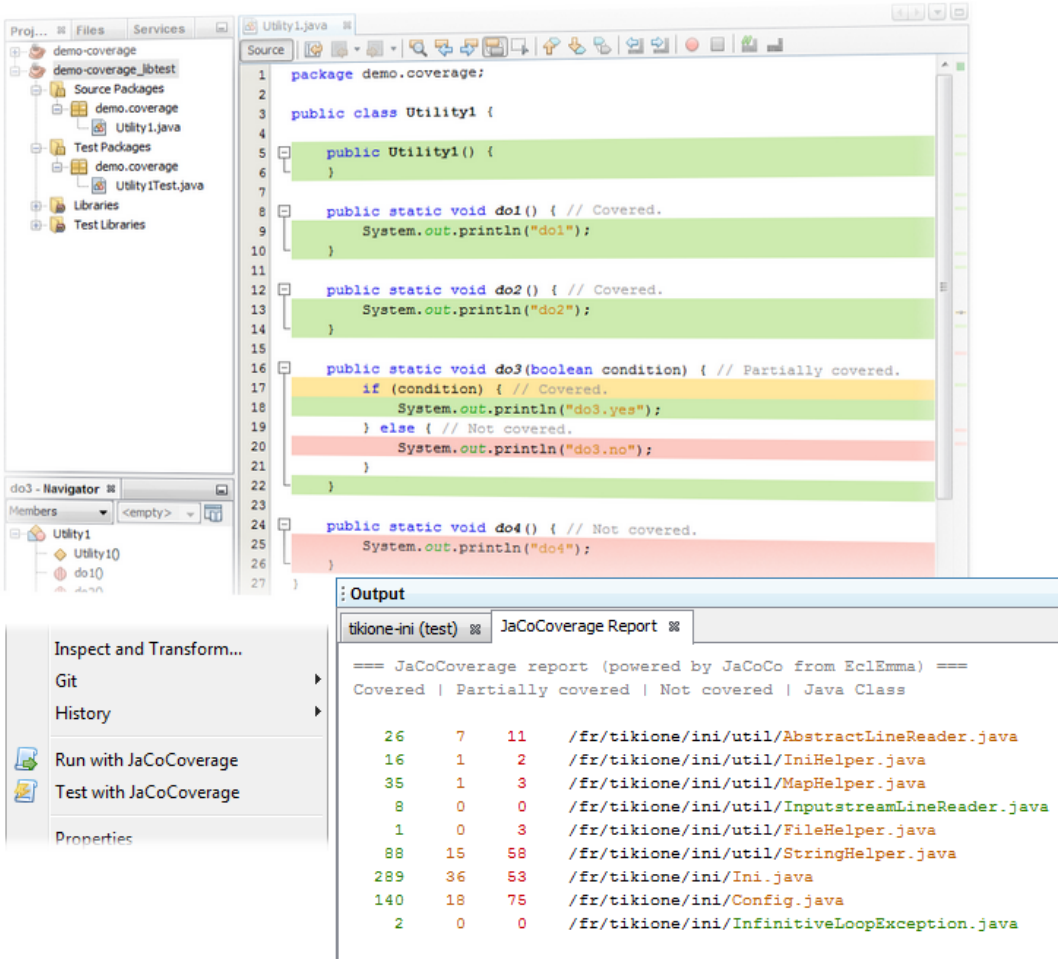
Druhým nejstahovanějším modulem pro Eclipse je eCobertura[2]. Poskytuje stejnou funkcionalitu jako EclEmma. Využívá Jacoco.



Obrázek 2: Modul EclEmma pro Eclipse

2.3 TikiOne JaCoCoCoverage

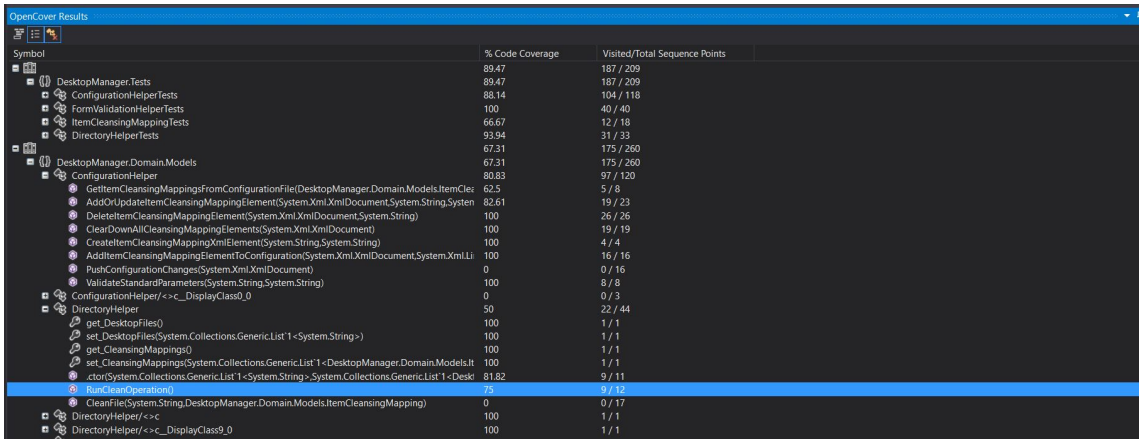
Nejstahovanějším modulem pro IDE NetBeans je TikiOne JaCoCoCoverage[3]. Využívá stejnou knihovnu pro záznam jako „EclEmma“, takže poskytuje stejnou funkcionalitu.



Obrázek 3: Modul TikiOne pro NetBeans

2.4 OpenCover

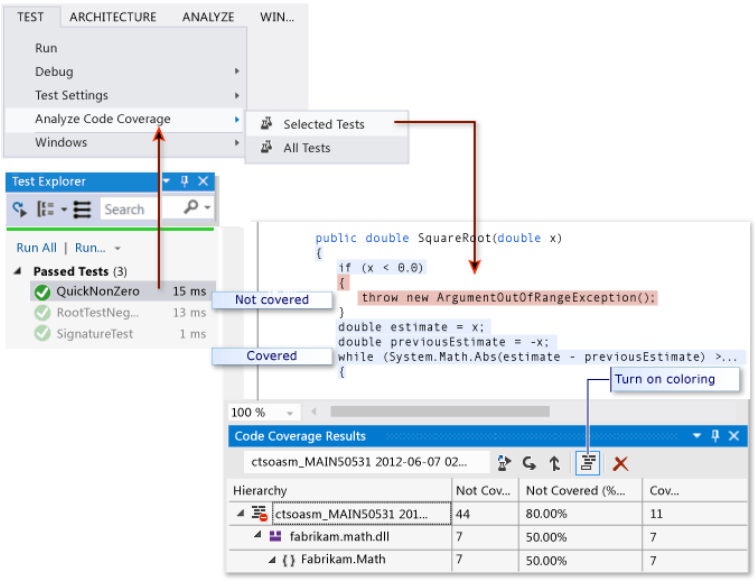
Volně dostupný modul do Visual studia pro C# (.NET) je OpenCover. Jako ostatní moduly zobrazí stromovou strukturu a zobrazí kolik procent kódu se provedlo. Využívá unit testy.



Obrázek 4: Modul OpenCover pro Visual studio

2.5 Visual studio

Visual studio obsahuje již modul pro pokrytí kódu. Poskytuje stejnou funkcionalitu jako „OpenCover“.

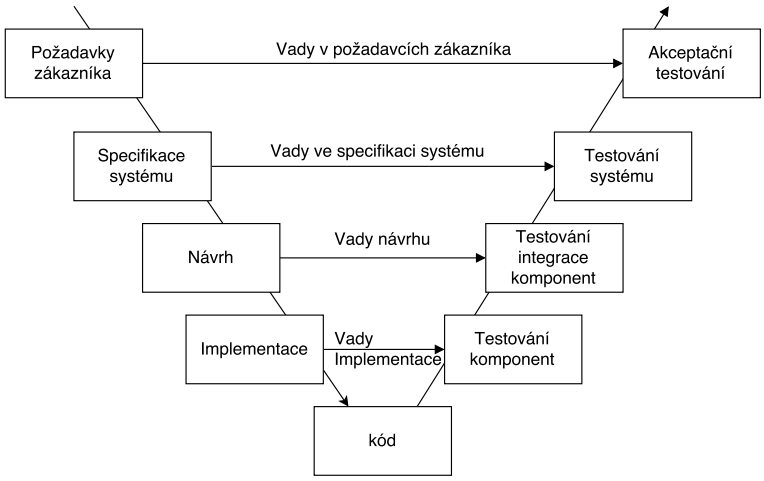


Obrázek 5: Modul ve Visual studiu

3 Úvod do testování softwaru

3.1 Testování softwaru

Testování softwaru má za úkol najít v testovaném softwaru chybu. Má předejít k vysokým nákladům když už je software nasazen. Pro testování se využívá tzv. „v-model“(viz. 6).



Obrázek 6: V-model testování

Pro každou fázi vývoje SW existuje i fáze testování.

- *Testování komponent* - nebo-li tzv. unitové testování. Testování tříd/metod.
- *Testování integrita komponent* - Testování komunikace mezi komponentami
- *Testování systému* - Testování celého systému
- *Akceptační testování* - Testování provádí zákazník, zda výsledný produkt je podle jeho představ

3.2 Metody testování

Metody testování se rozdělují na dva typy, černá skříňka(black box) a bílá skříňka(white box). Černá skříňka testuje kód jestli funguje na základě požadavků a bílá testuje samotný kód. Do černé skříňky patří například:

- Ekvivalentní rozdělení
- Analýza hraničních hodnot
- Testování přechodu mezi stavy

Mezi bílé skříňky paří například:

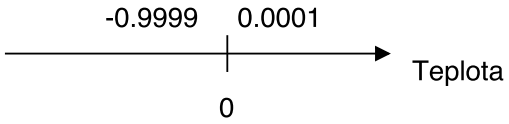
- Testování výrazů
- Testování skoků
- Testování pokrytí cest

Ekvivalentní rozdělení rozděluje vstup na podmnožiny, kdy systém reaguje stejně na vstupy z dané podmnožiny(příklad na obrázku 7).

(∞,0)	<0,100)	<100,∞)
Pevná látka	Kapalina	Pára

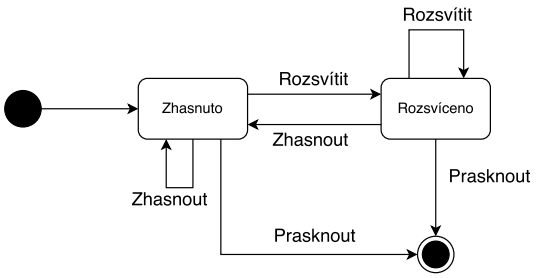
Obrázek 7: Ekvivalentní roděleni

Analýza hraničních hodnot testuje takové vstupy, které jsou co nejblíže k hranicím mezi dvěma podmnožinami(příklad na obrázku 8).



Obrázek 8: Hraniční rozdělení

Testování přechodu mezi stavy je založen na stavovém automatu(příklad na obrázku 9).

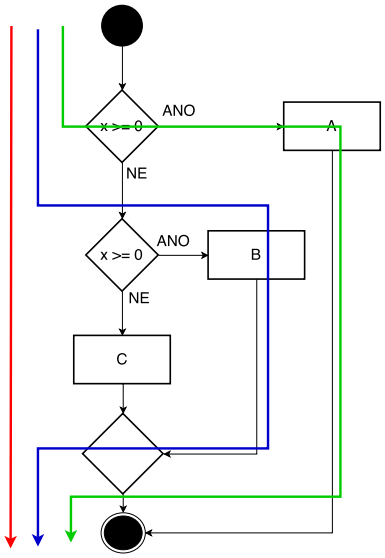


Obrázek 9: Testování přřechodu mezi stavy

Testování výrazů se testuje zda daný výraz se aspoň jednou provedl.
Pro výpočet pokrytí výrazů:

$$pokryti\ výrazů = Provedené\ výrazy/Celkový\ počet\ výrazů;$$

Testování skoků(rozhodování) testuje, za každé rozhodnutí se provedlo jak *false* tak i *true*.

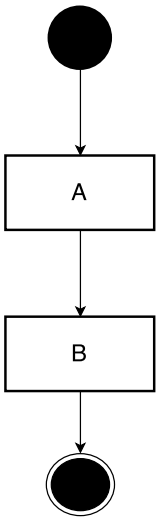


Obrázek 10: Testování skoků/rozhodování

Pro výpočet pokrytí skoků:

$$pokryti\ skoků = Provedené\ výstupy\ rozhodnutí/2 * Celkový\ počet\ rozhodnutí;$$

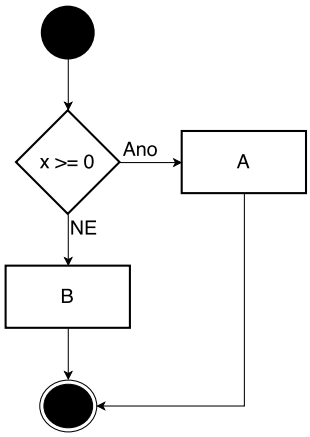
Testování pokrytí cest testuje, zda se prošly možné průchody kódem (viz. obrázek 10). Pro výpočet celkového pokrytí existují dva vzorce. Záleží jak dané rozhodnutí jsou situované.



Obrázek 11: Pokrytí cest - Kód za sebou

V prvním případě(obrázek 11) pokud jsou za sebou využije se vzorec:

$$pokryti\ cest = A \times B;$$



Obrázek 12: Pokrytí cest - Větvený kód

Pokud se kód větví(obrázek 12), tak se využije vzorec:

$$pokryti\ cest = A + B;$$

3.3 JUnit

Pro fázi „Testování kompoment“ (nebo také „unit testování“) existuje framework JUnit pro testování Java kódů. Využívá anotace kódu. Stačí danou metodu označit anotací „@Test“ a pustit danou třídu jako JUnit(viz. příklad 1).

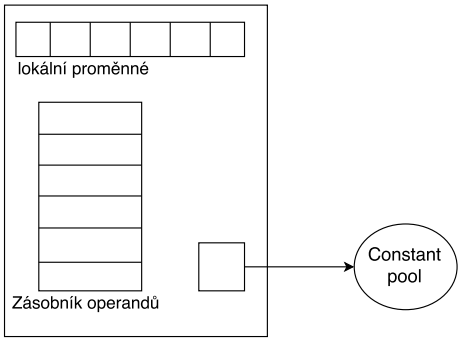
```
package test;

import static org.junit.Assert.assertTrue;

import org.junit.Test;

public class MyTest {

   @Test
   public void test3() {
      boolean ret = computer(0);
      assertTrue(ret);
   }
}
```

Výpis 1: Ukázka JUnit

4 Java technologie

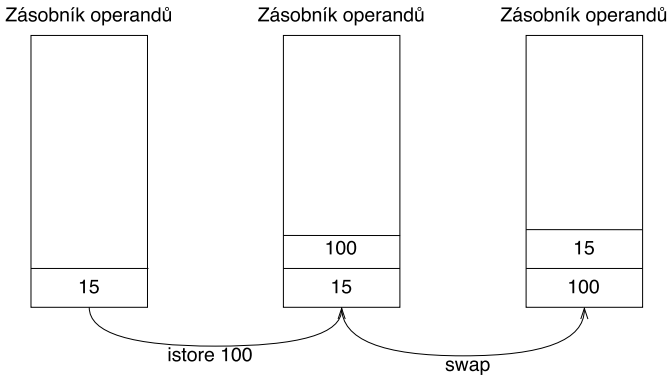
4.1 Java virtual machine

Java virtual machine (dále JVM [18, 19]) je virtuální stroj, který umožňuje spouštění java programů. Pro spuštění java programů je potřeba tzv. „Java bytecode“ (v češtině možno najít pod názvem mezikód nebo bajtkód). Tento kód se následně zkompiluje do strojového kódu. Existují v JVM dvě varianty kdy překládat do strojového kódy. Prvním a starším je tzv. „Just-In-Time“ (JIT), který přeloží bytecode buď při spuštění aplikace a to celý a nebo jenom část, která se má spustit. Druhým a rychlejším je tzv. „HotSpot“. Obsahuje i JIT. HotSpot optimalizuje ty metody, které jsou nejvíce používané. JVM je založen na zásobníkové architektuře. To znamená, že operandy instrukcí jsou uloženy na zásobníku, nikoliv v registrech. Každé vlákno má zásobník, kde si ukládá tzv. rámec (Frame)(struktura na obrázku 13). Rámec se skládá:



Obrázek 13: Rámec

- **Constant pool** - Reference na třídní seznam konstant. Ukládá všechny reference na proměnné a metody. Nejsou uloženy jako reference na určité místo v paměti, ale jako text typové signatury a jsou identifikovatelné číslem. Například „(ILjava/lang/String;)J“, značí, že daná metoda má dva parametry, „integer(I)“, „String(Ljava/lang/String;)“ a návratová hodnota je „long(J)“.
- **Lokální proměnné** - Pole kde jsou uloženy všechny lokální proměnné, které jsou využity v metodě.
- **Zásobník operandů** - Zásobník, kde se ukládají operandy instrukcí.



Obrázek 14: Příklad práce se zásobníkem operadů

4.2 Java bytecode

Java bytecode je mezikód, který se vytváří při kompilaci Java kódu. Tohle zajišťuje přenositelnost, ale jenom pokud na danou platformu existuje JVM, který bytecode spustí. Bytecode se skládá z posloupností instrukcí (tzv. „opcodes“ [20]). Každá instrukce se může skládat z nula nebo více operandů. Některé instrukce mají prefixy a sufixy. Prefixy určují s jakým typem proměnné pracuje. Například *astore* pracuje s objekty a *istore* pracuje s celočíselnými čísly (Integer). Sufix je ve tvaru _n, kde za n dosadí číslo. Maximální číslo záleží na instrukci. S čím instrukce pracuje a nebo jakou metodu volá se určuje pomocí constant pool. Každá metoda má pak odkaz na určitý záznam v constant pool pomocí čísla. Dekompilátory pak tohle číslo vypíšou ve tvaru „#N“, kde N je číslo záznamu.

5 Návrh a implementace

5.1 Knihovna Jacoco

Pro zjednodušení je využívána knihovna *Jacoco* (Java Code Coverage Library). Tuhle knihovnu využívají skoro všechny moduly pro pokrytí kódu v programovacím jazyku Java. Pomocí úpravy java bytecode získává data, která indikují, jestli daná instrukce se provedla. Obsahuje i algoritmus, který data převede na jednotlivé řádky kódu. Tyto informace pak využívají jednotlivé moduly pro vykreslení a vypočítání statistik o pokrytí.

Jacoco využívá tzv. „Java agent“, který byl přidán v Java 5. Je to v podstatě modul do JVM, který se spouští před vykonání samotného programu. Ten následně může pozměnit class soubory, které jsou nahrány do JVM. Pro spuštění programu s java agentem se zadá jako parametr ve tvaru „-javaagent:*cesta/k/agentovi.jar*“. Dále akceptuje dalších 14 parametrů ve tvaru „-javaagent:*cesta/k/agentovi.jar*=[parametr1]=[hodnota1],[parametr2]=[hodnota2]“. Nejdůležitější parametry jsou:

- **destfile** - cesta, kde se uloží výstupní soubor s daty
- **output** - určuje jakým způsobem se mají data uložit
  - **file** - data se uloží jako soubor. Cesta je definována v *destfile*
  - **tcpserver** - agent vytvoří TCP server. Klient se na něj následně připojí a příjme data
  - **tcpclient** - agent se připojí k serveru.
  - **none** - agent nemá vytvořit žádný výstupní soubor
- **address** - IP adresa nebo „hostname“ na který server se má připojit
- **port** - Port na který se má připojit a nebo na kterém má naslouchat v případě *tcpserver*

Dále obsahuje parametry, které určují, jaké třídy má analyzovat a které ne.

Jacoco se dá využít i bez agenta(viz. kód **??**). K tomu je potřeba jádro Jacoco, který agent obsahuje. Po stažení archivu z oficiálních stránek[5] obsahuje jak agenta tak i jádro. Pomocí třídy *Intrumenter* a metody *intrument*, které se vloží jako parametr *InputStream* třídy, kterou chce otestovat. Metoda následně upraví třídu tím, že vloží do metody vlastní kód. Tenhle postup pomáhá při debugování.

```
final String targetName = TestTarget.class.getName();
// LoggerRuntime ke sberu dat
final IRuntime runtime = new LoggerRuntime();
// Vlozi vlastni bytecode
final Instrumenter instr = new Instrumenter(runtime);
```

```
final byte[] instrumented = instr.instrument( getTargetClass(targetName),
    targetName);
//Spusteni runtime pro vykonani tridy
final RuntimeData data = new RuntimeData();
runtime.startup(data);
// Konvertovani pole bytu tridy na Class
final MemoryClassLoader memoryClassLoader = new MemoryClassLoader();
memoryClassLoader.addDefinition(targetName, instrumented);
final Class<?> targetClass = memoryClassLoader.loadClass(targetName);
if(DEBUG) {
   return;
}
// Spusteni jiz tridy, kterou chceme vykonat
final Runnable targetInstance = (Runnable) targetClass.newInstance();
targetInstance.run();
```

Výpis 2: Ukázka java kódu pro prezentaci výsledných dat Jacoco

Výsledná data jsou obsaženy ve třídě *ExecutionDataStore*. Ta obsahuje mapu, kde ke každé třídě, která byla analyzována, obsahuje třídu *ExecutionData*. Tato třída obsahuje již data, které agent shromáždil. Data jsou uloženy v poli *boolean*, tzv. „probe“. Pokud je hodnota *true*, daná instrukce(nebo posloupnost instrukcí) se provedla a pokud *false*, tak se daná instrukce neprovedla.

```
public static class TestTarget implements Runnable {

   private String hello = "hello";

   public void run() {
      compute(0);
      System.out.println();
   }

   public void compute(int a) {
      System.out.println(hello);
      for(int i = 0; i < a; i++) {
         System.out.println();
      }
      System.out.println();
   }
}
```

Výpis 3: Ukázka java kódu pro prezentaci výsledných dat Jacoco

Výstup, který Jacoco vrátí v podobně pole v případně kódu 3, bude: *[true, true, true, true, false, true, true]*. Pokud sami kód projdeme, dojdeme k tomu, že jediný kód, který se neprovede je *println* uvnitř cyklu. V poli najdeme jedinou hodnotu *false* na pozici 4.

Jak je vidět, tak zpětná analýza je velmi pracná. Nevíme přesně které data odpovídají k daným kódům (řádku). Proto Jacoco poskytuje funkci, která převede pole *boolean* na dané řádky. A to pomocí třídy *Analyzer*, která poskytuje metodu *analyzeClass*. Dále poskytuji i metody, které analyzují ty třídy, které se nacházejí v dané složce. Metodě předáme dva parametry. Prví je třída *ExecutionDataStore*, kterou nám poslal agent. A druhým parametrem je *CoverageBuilder* ve které budou po analýze výsledná data. Z ní pak získáme kolekci všech tříd, které analyzovala. Data jsou uložena ve třídě, která implementuje rozhraní *IClassCoverage*. Pro výpis všech řádků, zda se provedly využijeme kód[4], který si vytáhne nejnižší a nejvyšší číslo řádku. To potom pomocí metody *getLine* a následně *getStatus* získáme, zda se daný kód provedl(viz. kód 4). Vrací 4 hodnoty.

- **0** - Na daném řádku není kód
- **1** - Kód se neprovedl
- **2** - Kód se provedl. V případě větvení kódu, se provedly všechny možnosti
- **3** - U větvení kódu se neprovedly všechny větve kódu

```
for (int i = cc.getFirstLine(); i <= cc.getLastLine(); i++) {
  System.out.printf("Line %s: %s\n", Integer.valueOf(i), cc.getLine(i).
      getStatus());
}
```

Výpis 4: Získání pokrytí kódu z IClassCoverage

```
Instukce: 3/28
Větve: 1/2
Řádků: 1/10
Metody: 0/3
Složitost: 1/4
Line 0: 2
Line 1: 0
Line 2: 2
Line 3: 0
Line 4: 0
Line 5: 2
Line 6: 2
Line 7: 2
Line 8: 0
Line 9: 0
Line 10: 2
Line 11: 3
Line 12: 1
Line 13: 0
Line 14: 2
Line 15: 2
```

Obrázek 15: Výsledky analýzy kódu [3]

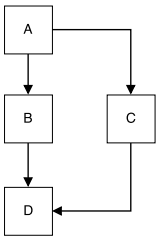
Podle výsledků analýzy na obrázku 15 je vidět, že na 11. řádku (cyklus *for*) se neprovedly všechny větve. Program nevkročil do cyklu.

Analýza dále poskytuje i menší statistiky. Ty jsou vidět na prvních 5 řádcích na obrázku 15. První je kolik se instrukcí neprovedlo z celkového počtu. Druhý je kolik větví se neprovedlo. Třetí je kolik řádků se neprovedlo. Čtvrtým je kolik metod se neprovedlo a pátým je tvz. cyklomatická složitost. Ta vypočítá kolik je možných cest grafem(kódem) a vypočítá se vzorcem:

$$M = E - N + 2 \times P;$$

kde:

- **M** - cyklomatická složitost
- **E** - počet hran v grafu
- **N** - počet uzlů v grafu
- **P** - počet připojených komponent



Obrázek 16: Příklad grafu pro cyklomatickou složitost

Pro graf na obrázku 16 se vypočítá cyklomatická složitost tak, že počet hran(E) je 4, počet uzlů(N) je 4 a počet připojených komponent(P) je 1. Takže po dosazení do vzorce je výsledná složitost 2.

Tyhle údaje následně využívají moduly k vypsání statistik ohledně testovaného kódu.

5.2 Úprava Jacoco

Podle předešlé kapitole(5.1) je vidět, že Jacoco umožňuje vytvořit statistiku jenom o pokrytí instrukcí(řádcích). Proto je potřeba jej upravit pro potřeby téhle diplomové práci.

5.2.1 Kompilování a zabalení knihovny

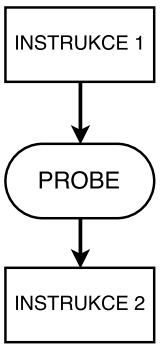
Ke kompilování a zabalení knihovny se využívá nástroj Maven. Nejprve se musí zkompilovat zabalit jádro(core) knihovny. To se provede příkazem „mvn install“ v adresáři „org.jacoco.core“. Kvůli tomu, že další části projektu využívají jádro a zabalenou knihovnu berou z lokálního repozitáře, musí se provést akce „isntall“. Jako další se zabalí projekt uvorg.jacoco.agent.rt. Použije se stejný příkaz, ale v adresáři projektu. A jako poslední se zabalí projekt „org.jacoco.agent“. Může se použít stejný příkaz a nebo „mvn clean package“.

První příkaz vytvoří zabalenou knihovnu, která se následně importuje do projektu. Třetí příkaz vytvoří agenta.

5.2.2 Princip úpravy testovaného kódu

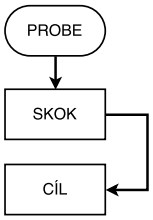
Jacoco neupravuje Java kód, ale až na straně JVM, kde upraví Java bytecode. Jacoco vkládá svůj vlastní kód, který označí, že daná instrukce(nebo posloupnost instrukcí) se provedla. Tento kód(tzv. „probe“) uloží do pole *true*, že daná instrukce se provedla. Pokud by se ukládal „probe“ u každé instrukce, zpomalovalo by to samotný program. Proto Jacoco vkládá čtyřmi způsoby do kódu vlastní kód.

První(obrázek 17) je, že mezi dvěma instrukcemi se vloží kód(„probe“). Zde je zapotřebí dát pozor, protože tím se nemyslí jednoduché instrukce jako inicializace nové lokální proměnné, vložení hodnoty do proměnné a nebo inkrementace nebo dekrementace.



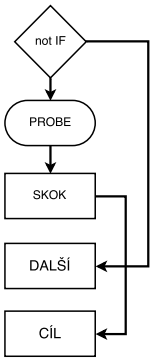
Obrázek 17: První způsob vložení „probe“

Druhý(obrázek 18) způsob vloží kód před nepodmíněný skok(instrukce *GOTO*).



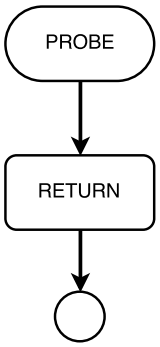
Obrázek 18: Druhý způsob vložení „probe“

Třetí(obrázek 19) způsob vkládá kód do podmíněného skoku(instrukce *IF..*). Tento způsob je o něco složitější. Jacoco podmínku invertuje a přidá kód do bloku, když podmínka je *false*. Po úpravě podmínky je tento kód v bloku *true*.



Obrázek 19: Třetí způsob vložení „probe“

Čtvrtý(obrázek 20) způsob vkládá kód před instrukcí, která opouští metodu. To jsou instrukce *xRETURN* a *THROW*



Obrázek 20: Čtvrtý způsob vložení „probe“

Jak bylo již napsáno, Jacoco nevkládá svůj kód všude mezi jednotlivými kódy. A to může vyústit ve špatnou analýzu kódu. Například v kódu 5 jde vidět, že při inicializaci nové proměnné vyhodí výjimku, protože se dělí nulou. Jelikož mezi prvním a druhým řádkem není vložen kód pro detekci, tak se špatně vyhodnotí, že oba dva kódy se neprovedly.

```
System.out.println("Zdravim");
int tmp = 10/0;
```

Výpis 5: Chybná analýza kódu

Největší podmínkou je, že vložený kód(probe) nesmí nijak ovlivnit původní kód. To znamená, že před a po musí být zásobník operandů stejný.

Vložený kód je velmi jednoduchý a proto je i velmi efektivní. Využívá pro jednoduchost lokální proměnou v rámci metody, kde uchovává pole, které pak upravuje. Skládá se ze čtyř instrukcí. První je načtení lokální proměnné do zásobníku. Druhá je vložení čísla indexu do zásobníku. Dále vložení do zásobníku číslo 1(neexistuje instrukce speciálně pro vložení *boolean* typu) a jako poslední je vložení instrukce, která vloží na daný index hodnotu 1. Pro větší optimalizaci Jacoco vytvoří pro každou třídu statickou metodu(viz. kód 6 , kde inicializuje statickou třídní

proměnou, kde se ukládají data o pokrytí. Metoda pak danou proměnnou vrací, kde si jí daná metoda uloží do lokální proměnné.

```
private static /* synthetic */ boolean[] $jacocoInit() {
   boolean[] arrbl = $jacocoData;
    if (arrbl == null) {
      Object[] arrobject = new Object[]{3835393719500207983L, "main/
          CoreTutorial$TestTarget", 3};
        Logger.getLogger("jacoco-runtime").log(Level.INFO, "3d4eac69",
            arrobject);
        arrbl = CoreTutorial.TestTarget.$jacocoData = (boolean[])arrobject[0];
    }
    return arrbl;
}
```

Výpis 6: Inicializační metoda

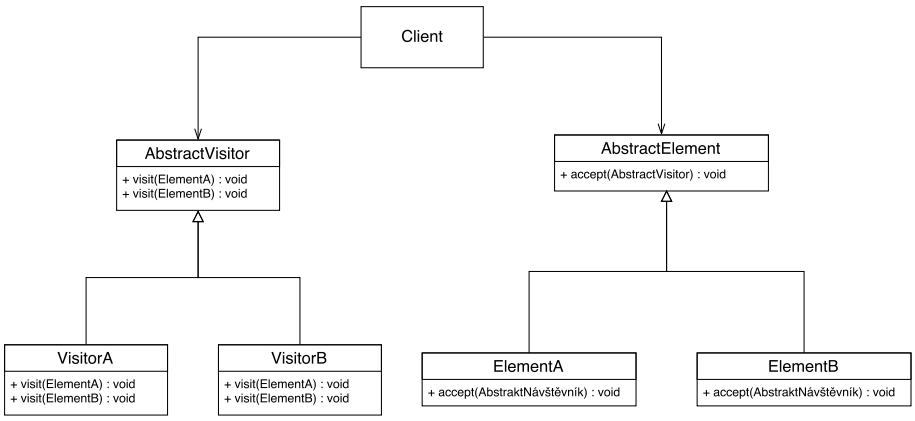
Jacoco využívá principu logování k tomu, aby se dostal k získaným datům. Vytvoří „handler“, který následně zpracování zprávy, které „loggeru“ předáváme. Princip funguje na zajímavém způsobu. „Loggeru“ předáme tři parametry, kde první je identifikátor třídy. Druhý parametr je úplný název třídy a třetí je celkový počet probe, který je v dané metodě. „Handler“ následně tyhle informace uloží a k názvu třídy inicializuje pole typu *boolean*. Jelikož „handler“ pracuje s parametry které jsou uloženy v poli typu *Object*, tak se na místo původního identifikátoru třídy vloží nově incializované pole. To následně už v dané třídě se uloží do třídní statické proměnné(7. řádek v kódu 6). „Logger“ se používá v případě upravování bez agenta. V případě využívání agenta se využívá stejný princip, kdy se zamění objekty, ale přístup k datům je jiný.

Pro dekompilováni bytecodu se využíval online dekompilátor na stránkách javadecompilers.com. Na stránkách se dá vybrat z několika dekompilátorů. Velmi rychlý a s podporou Java 8 je CFR a který jako jediný dokázal kódy přeložit skoro dobře. Kvůli výpadku stránek, který trval okolo měsíce, se využíval přímo program CFR. Ovládá se přes příkazovou řádku. Pro jednoduché dekompilováni stačí jako parametr zadat class soubor a nebo jar soubor. Jako výchozí nastavení se výstup vypíše do konzole, ale při výstup do souboru se zadá parametr „–outputdir“. Jak je vidět na kódu 6, tak dekompilováni se neprovedlo úplně správně. Vytvořené pole se má uložit do proměnné *$jacocoData*, ale CFR vytvořilo novou proměnnou *arrbl*. Zde je pak patrné, že v další metodě se znova provede nová inicializace pole, protože proměnná *$jacocoData* bude *null*. Pro zobrazení přímo java bytecode se využíval program Java Bytocode Editor(JBE). Ten umožňuje i editaci. Dokáže zobrazit všechny informace o třídě(Constant pool, lokální tabulku proměnných). Zde je zajímavé, že nezobrazuje aktualizovanou lokální tabulku proměnných. Dokáže ale zobrazit vložené třídní proměnné, kterou tam Jacoco vkládá.

5.2.3 Vkládání bytecodu

Pro analýzu a manipulaci se využívá framework ASM. Tento framework je velice populární, jelikož velmi usnadňuje práci s java bytecode a je velmi rychlý.

Pro manipulaci se využívá podobný vzor jako je Visitor(obrázek 21). Ten umožňuje lepší práci s java bytecode. Programátor tak je celkově odstřižen od bajtové struktury bytecode a ostatních vlastností bytecode.

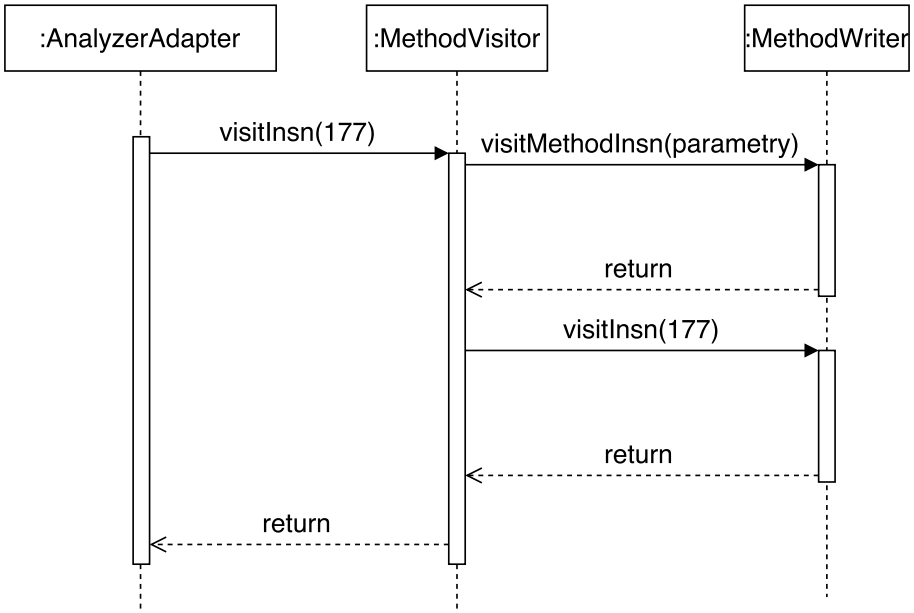


Obrázek 21: Návrhový vzor Visitor

Element je zastoupen v ASM jako *ClassReader* a jako návštěvník jako *ClassVisitor*. Jacoco si vytvoří svého vlastního návštěvníka. Třída se jmenuje *ClassProbesAdapter* která dědí z *ClassVisitor*. *ClassReader* následně prochází celou strukturu class souboru. Samotná struktura, která prochází jednotlivé údaje o třídě(proměnné, metody, anotace a podobně) je velmi složitá. Nejdůležitější je, ale třída *MethodVisitor*. Ta prochází samotný bytecode a ve kterém následně můžeme vkládat svůj vlastní java bytecode.

Samotné vkládání instrukcí funguje na stejném principu jako celý framework. Navštívíme tu instrukci, kterou chceme uložit. To se provádí zavolání metody *visit*, která se nachází ve třídě *MethodWriter*. Například pokud chceme vložit instrukci, která uloží proměnou typu *boolean* do pole, zavoláme metodu *visitInsn*. Té předáme parametr číslo instrukce kterou má vložit. V tomhle případě číslo 84. Pro jednoduchost existuje třída *Opcodes*, která obsahuje statické proměnné všech instrukcí. Takže nemusíme znát čísla instrukcí, ale vyhledáme název instrukce, kterou chce uložit. V tomhle případně vyhledáme jméno instrukce *bastore*. Názvy metod *visit* vždy závisí na povaze instrukce. Pro volání metod existuje metoda *visitMethodInsn*, které předáme o jakou metodu se jedná(statická, metoda z rozhraní nebo klasická). Dále jí předáme další čtyři parametry. První je v jaké třídě se metoda nachází. Druhý parametr je název metody. Třetí parametr je signatura metody a poslední je zda je vlastníkem metody je rozhraní. Poslední parametr se přidal kvůli nové funkci v Java 8. V Java 8 se nyní může přidat statická metoda do rozhraní a proto je potřeba označit, zda daná metoda je v rozhraní.

Na obrázku 22 je jednoduchý sekvenční digram pro vložení vlastní instrukce pro volání metody před instrukci 177. Instrukce s číslem 177 se jmenuje *return*, která opouští metodu. Zde chceme zavolat určitou metodu před opuštěním metody. Jak bylo již řečeno, struktura pro analýzu a následnou manipulaci bytecodu je velmi složitá, proto byl sekvenční diagram zjednodušený. Dále pro zjednodušení diagramu chybí podmínka v *MethodVisitor* zda navštívená instrukce je *return*. Jelikož chceme aby naše volání metody bylo před opuštění metody, tak se zavolá metoda *visitMethodInsn* s danými parametry (viz. předešlý odstavec) na třídu *MethodWriter*. Tak vloží naší novou instrukci a dále za ní zavoláme metodu *visitInsn*, která vloží již původní instrukci která tam byla. A to je instrukce *return*.



Obrázek 22: Sekvenční diagram pro vložení vlastní instrukce

5.2.4 Princip zjištění průchodu kódu

Jak již bylo zmíněno, tak Jacoco dokáže vyhodnotit jenom pokrytí kódu. Proto musel být vymyšlen princip, jak dokázat zachytit, jak se daný kód vyhodnotil. Asi nejlepším způsobem jak toho dosáhnout je ukládání časových značek (tzv. „timestamp“). Obecně je to sekvence znaků, které označují přesný čas vzniku nějaké události. V tomhle případě to je číslo které určuje, v jakém čase se to provedlo. Takže pokud si uložíme čas kdy se daný kód provedl, tak poznáme pak následnou analýzou časových značek, jak se daný kód provedl. Jelikož počítače pracují v řádech nanosekund, musí se použit čas v řádech nanosekund. V jazyku Java se dá využít metoda *nanoTime* ve třídě *System*. Ta poskytuje dostatečnou přesnost.

Pro každý probe se ukládá kolekce časových značek. Pro jednoduchost se ukládají do dynamického pole a ne do obyčejného pole. Je to z toho důvodu, že vkládaný kód, který zjistí časovou

značku, nemusí být složitý. Je potřeba dynamickou velikost pole a kolekce(*ArrayList*) to dělají za programátory samy.

Jelikož kód obsahuje více probe musí se ukládat pro každou probe vlastní časové značky. Proto se využívá hashovací tabulka, kde klíč je číslo probe a hodnota je pole(*ArrayList*) časových značek.

5.2.5 Inicializace proměnné

Pro inicializaci potřebného atributu se využívá již napsaná inicializační metoda s názvem *$jacocoInit*(viz. kapitola 5.2.2). Nejprve je ale zapotřebí vytvořit atribut, který uchovává hashovací tabulku s časovými značkami. Jacoco vytváří svůj atribut v metodě *createDataField* ve třídě *ClassFieldProbeArrayStrategy* a ve třídě *InterfaceFieldProbeArrayStrategy*. Jelikož od Java 8 může rozhraní obsahovat i tělo statické metody musí se zpracovávat i rozhraní. Proto musí existovat dvě „strategie“, které ukládají vlastní kód. První třída upravuje třídy a druhá třída upravuje rozhraní. K vytvoření atributu slouží metoda *visitField*(viz. kód 7), která má 5 parametrů. První je modifikátor přístupu, syntetický atribut a nebo zda je „Deprecated“. Tyto vlastnosti se následně spojují pomocí operátoru /. Tyto vlastnosti pro jednoduchost se nacházejí ve třídě *Opcodes*. Druhý parametrem je název atributu. Třetí parametr typ proměnné. Čtvrtý parametr je signatura a poslední je počáteční hodnota atributu. Veškeré textové data jsou ve třídě *InstrSupport*. Atribut označíme, že je uměle vytvořen. K tomu slouží modifikátor *synthetic* a dále jej označíme aby při případné serializaci třídy se atribut taky neuložil. K tomu slouží modifikátor *transient*. Atribut je pojmenován „$timeStampData“. Do signatury vložíme typy generik. Kde klíč je *Integer* a hodnota je *List*.

```
cv.visitField(Opcodes.ACC_SYNTHETIC | Opcodes.ACC_PRIVATE | Opcodes.ACC_STATIC
    | Opcodes.ACC_TRANSIENT, "$timeStampData", "Ljava/util/Map;", "Ljava/util/
    Map<Ljava/lang/Integer;Ljava/util/List;>;", null);
```

Výpis 7: Výsledný kód pro vložení atributu do třídy

```
private static transient /* synthetic */ Map<Integer, List> $timeStampData;
```

Výpis 8: Výsledný atribut pro ukládání časových značek

Samotná inicializace atributu se neprovádí ve vloženém kódu. Ta se provádí při vytváření nové instance *ExecutionData*. Ta obsahuje identifikátor třídy, název třídy, probe tak i časové značky k daným probe. Jelikož se inicializační metoda vytváří až po vkládání kódu pro sběr dat, tak se už ví kolik celkově je potřeba polí pro uchovávání časových značek. To jak se dostaneme k těmhle datům závisí jak byl program spuštěn. Pokud bez agenta, tak se využívá logování a pokud s agentem využívá se metoda *equals* ve třídě *RuntimeData*.

V prvním případě(viz. kód 9) v metodě *publish* získáme data z *LogRecord*. Následně je předáme *RuntimeData*, kde se vytvoří instance a zamění se.

```
@Override
public void publish(final LogRecord record) {
   final Long classid = (Long) record.getParameters()[0];
   if (key.equals(record.getMessage())) {
      data.getProbes(record.getParameters());
      data.getTimestamp(record.getParameters(), classid);
   }
}
```

Výpis 9: Získání instance pole bez agenta

V druhém případě(viz. kód 10) data získáme v metodě *equals*. Ty jsou obsaženy přímo jako parametr v poli objektů. A dál to probíhá stejně jako v předchozím případě.

```
@Override
public boolean equals(final Object args) {
   if (args instanceof Object[]) {
      final Object[] objs = (Object[]) args;
      // System.out.println("baf, jsem v equals");
      final Long classid = (Long) objs[0];
      getProbes(objs);
      getTimestamp(objs, classid);
   }
   return super.equals(args);
}
```

Výpis 10: Získání instance pole s agentem

Ve třídě *RuntimeData* pro prohození parametrů(viz. kód 11) se využívá metoda *getTimestamp*, které předáme pole objektů(získáné z předešlého kroku) a identifikátor třídy. V této metodě následně získáme *ExecutionData* dané třídy a z ní pak kolekci časových značek. Tu následně vložíme na druhé místo v poli objektů.

```
public void getTimestamp(final Object[] args, final Long classid) {
   final ExecutionData ed = store.get(classid.longValue());
   args[1] = ed.getMapOfTimestamp();
}
```

Výpis 11: Záměna objektu za kolekci časových značek

V předešlém kroku se kolekce časových značek vložila na druhé místo v poli objektů. Tuhle kolekci si uložíme do třídní proměnné ve třídě kterou upravujeme(viz. kód 12). Pro tuhle operaci je potřeba čtyři bytecode instrukce. Nejprve vložíme do zásobníku číslo jedna pomocí instrukce

iconst_1, jelikož chceme vytáhnout objekt druhé pozici v poli. Dále pomocí instrukce *aaload* vytáhneme objekt z pole. Jelikož je to pole objektů, musí se objekt konvertovat na *Map* a to pomocí instrukce *checkcast*(využití metody *visitTypeInsn*), které ještě předáme typ na který to chceme překonvertovat. Poslední instrukce je vložení kolekce do třídní proměnné pomocí instrukce *putstatic*.

```
mv.visitInsn(Opcodes.ICONST_1);
mv.visitInsn(Opcodes.AALOAD);
mv.visitTypeInsn(Opcodes.CHECKCAST, "java/util/Map");
mv.visitFieldInsn(Opcodes.PUTSTATIC, className, "$timeStampData", "Ljava/util/
    Map;");
```

Výpis 12: Vložení kódu pro uložení kolekce do třídní proměnné

Dále jsou zapotřebí ještě udělat dva kroky. První krok je aktualizovat rámec před ukončení podmínky *if*(před metodu *visitLabel*). To se provádí zavolání metody *visitFrame*. Té se předá parametr zda chce parametry přiřadit ke stávajícím a nebo úplně nové. Jelikož zde nepoužíváme žádné lokální proměnné tak druhý parametr je nula a třetí je pole s nulovou délkou. Poslední dva parametry se zabývají, co se nachází na zásobníku. V tomhle případě se nachází kolekce *Map* a pole typu *boolean*. Druhý krok je určení maximální velikosti zásobníku a lokálních proměnných. To se provádí metodou *visitMaxs*. Zde je zapotřebí znát přesně co se děje na zásobníku aby se určila maximální délka zásobníku. Pro potřebu vložení kolekce do třídní proměnné je zapotřebí maximálně dvě pole. Proto následně v kódu je *size + 2*.(viz. kód 13)

```
if (withFrames) {
   mv.visitFrame(Opcodes.F_NEW, 0, FRAME_LOCALS_EMPTY, 2,
   FRAME_STACK_ARRZ);
}
mv.visitMaxs(Math.max(size + 2, 3), 0);
```

Výpis 13: Vložení kódu pro uložení kolekce do třídní proměnné

Po přeložení bytecodu do javy je vidět, že vložený kód(viz. kód 14) je správný a dekompilováni proběhlo správně. Dekompilováni nekontroluje rámce. To se děje až při načítání bytecode do JVM. Proto se může stát, že dekompilováni proběhlo v pořádku, ale následně při spuštění kódu to vyhodí výjimku, že je chyba v rámci.

```
private static /* synthetic */ boolean[] $jacocoInit() {
   Map map = $timeStampData;
    boolean[] arrbl = $jacocoData;
    if (arrbl == null) {
      Object[] arrobject = new Object[]{3835393719500207983L, "main/
          CoreTutorial$TestTarget", 3};
```

```
        Logger.getLogger("jacoco-runtime").log(Level.INFO, "41a4555e",
            arrobject);
        arrbl = CoreTutorial.TestTarget.$jacocoData = (boolean[])arrobject[0];
        $timeStampData = (Map)arrobject[1];
    }
    return arrbl;
}
```

Výpis 14: Upravená inicializační metoda

5.2.6 Kód pro sběr časových značek

Pro vkládání bytecodu, který sbírá data ohledně procházení kódu, se stará třída *ProbeInserter*. Ve třídě se nachází důležitý atribut *variable*. Ten určuje, na jaké pozici v poli lokálních proměnných se nachází pole pro ukládání probe. Jelikož využíváme svojí lokální proměnnou musíme jí nejprve do pole přidat. To se děje v metodě *visitFrame*. Metoda prochází jednotlivě položky v poli lokálních proměnných. Pokud narazí na pozici, kde se mají vložit vlastní proměnné, tak vloží jejich typ a další položky posune o dva indexy(vkládáme pole pro ukládání probe a kolekci pro ukládání časových značek). Atribut *variable* ukazuje na pozici, kde je uloženo pole pro vkládání probe. Kolekce pro ukládání časových značek je vložena hned na další pozici. Proměnná *newLocal* je pole, jak bude vypadat nové pole lokálních proměnných. (viz. kód 15)

```
if (pos == variable + 1) {
   newLocal[newIdx++] = "java/util/Map";
   pos++;
}
```

Výpis 15: Uprava metody visitFrame

O vkládání bytecodu se stará metoda *insertProbe*. Jako parametr se jí předává číslo probe. To značí, kolikátý probe se vkládá. Číslo určuje i pozici v poli. Pro uložení časové značky stačí jenom 9 instrukcí(viz. kód 16). Nejprve se načte kolekce do které se uloží časová značka. Parametr metody se využije pro načtení listu, který určuje o který probe se jedná. Jelikož klíč je typu *Integer* musí se převést *int*. K tomu se využívá statická metoda *valueOf* ve třídě *Integer*. Jelikož je metoda statická, tak parametr o jakou metodu se jedná je *INVOKESTATIC*. Další instrukce je pro získání hodnoty z hashovací tabulky. K tomu složí metoda *get*. Metoda *get* je, ale deklarovaná v rozhraní musí se použít parametr *INVOKEINTERFACE*. Jelikož metoda *get* vrací *Object*, musí se překonvertovat na *List*. Dále se zavolá metoda *nanoTime* ve třídě *System*. Ta vrací *long*. Jelikož kolekce pracuje s *Long* musí se překonvertovat. Následně se zavolá metoda *add*, která vloží číslo do listu. I zde je metoda *add* deklarována v rozhraní, proto se použije parametr *INVOKEINTERFACE*. Jako poslední se zavolá instrukce *pop*. Hlavní podmínka je aby stav

rámce byl stejný po vložení jako před vložení bytecodu. Proot se musí instrukcí *pop* odstranit operand který zůstal na zásobníku.

```
mv.visitVarInsn(Opcodes.ALOAD, variable + 1);
InstrSupport.push(mv, id);
mv.visitMethodInsn(Opcodes.INVOKESTATIC, "java/lang/Integer", "valueOf", "(I)
    Ljava/lang/Integer;", false);
mv.visitMethodInsn(Opcodes.INVOKEINTERFACE, "java/util/Map", "get", "(Ljava/
    lang/Object;)Ljava/lang/Object;", true);
mv.visitTypeInsn(Opcodes.CHECKCAST, "java/util/List");
mv.visitMethodInsn(Opcodes.INVOKESTATIC, "java/lang/System", "nanoTime",
"()J", false);
mv.visitMethodInsn(Opcodes.INVOKESTATIC, "java/lang/Long", "valueOf",
"(J)Ljava/lang/Long;", false);
mv.visitMethodInsn(Opcodes.INVOKEINTERFACE, "java/util/List", "add",
"(Ljava/lang/Object;)Z", true);
mv.visitInsn(Opcodes.POP);
```

Výpis 16: Vložení kódu pro uložení časové značky

Dekompilováni(kód 17) a následný test v podobě vykonání testovacího programu proběhne v pořádku.

```
((List)map.get(0)).add(System.nanoTime());
```

Výpis 17: Kód pro uložení časové značky

Dále je potřeba upravit metodu *visitMaxs* jelikož byl přidán nový bytecode do původního. Bytecode pro uložení časové značky vyžaduje 3 volné pozice na zásobníku operandů. Proto se přičítá tři k původní velikosti(plus tři, které vyžaduje pro uložení probe). Dále kvůli vložení nových proměnných do pole lokálních proměnných v metodě je přičtena jednička.

Kvůli změně pozic v poli lokálních proměnných se musí vždycky upravit index proměnné. To se děje pomocí metody *map*. Ta vždycky pokud je využívána proměnná, které se změnil index, přičte dvojka.

5.2.7 Úprava odesílání dat

Po skončení programu následně agent odešle nebo uloží(závisí na nastavení agenta) data které nasbíral. Proto bylo potřeba upravit odesílání daty, aby odeslal i časové značky jednotlivých tříd. To se provádí ve třídách *ExecutionDataReader* a *ExecutionDataWriter*. Nejprve se odešle speciální značka, která značí, že se začínají odesílat data. Dále se odešle identifikátor třídy, název třídy, pole probů a následně časové značky.

Při odesílání(obrázek 18) se využívá třída *CompactDataOutput*, která dědí ze třídy *DataOutputStream*, která pracuje se proudy(stream). Přidaly se jí metody, které ukládají složitější

struktury dat. Pro uložení časových značek se využívá metoda *writeTimestamps*. Kvůli neobjektovém zapisování do streamu, se musí jednotlivé časové značky odesílat postupně(obrázek 23). Nejprve se odešle počet probe pro danou třídu. Následují pak bloky pro jednotlivé proby. Kde blok obsahuje kolik časových značek obsahuje, index probu a pak jednotlivé časové značky.

Obrázek 23: Struktura odesílání časových značek

- **Velikost** - Počet probů
- **Počet** - Počet značek pro jeden probe
- **Index** - Index probu
- **značka** - Časová značka

```
public void writeTimestamps(final Map<Integer, List<Long>> map) throws
    IOException {
   if (map != null) {
      writeVarInt(map.keySet().size());
      for (final Integer i : map.keySet()) {
         if (map.get(i) != null) {
            writeVarInt(map.get(i).size());
            writeVarInt(i.intValue());
            for (final Long l : map.get(i)) {
               writeLong(l.longValue());
            }
         } else {
            writeVarInt(0);
            writeVarInt(i.intValue());
         }
      }
   } else {
   writeVarInt(0);
   }
}
```

Výpis 18: Odesílání časových značek

Pro přijímání dat se využívá třída *CompactDataInput*, která dědí ze třídy *DataInputStream*. Pomocí třídy *readTimestamps* se načítají jednotlivé časové značky pro daný probe(viz. kód 19).

```
public Map<Integer, List<Long>> readTimestamps() throws IOException {
   final Map<Integer, List<Long>> map = new HashMap<Integer, List<Long>>();
   final int countOfProbes = readVarInt();
   for (int i = 0; i < countOfProbes; i++) {
      final int countOfTimestamps = readVarInt();
      final int indexOfProbe = readVarInt();
      final List<Long> list = new ArrayList<Long>();
      if (countOfTimestamps > 0) {
         for (int z = 0; z < countOfTimestamps; z++) {
            list.add(new Long(readLong()));
         }
      }
      map.put(new Integer(indexOfProbe), list);
   }
   return map;
}
```

Výpis 19: Čtení časových značek

5.2.8 Úprava analyzátoru bytecodu

Jacoco obsahuje analyzátor, který převede jednotlivé probe na řádky. Celý princip funguje na převedení bytecode na třídu *Instruction*, která reprezentuje instrukci. Ta si uchovává informace o jaký řádek se jedná, zda byla provedena, referenci na předchůdce a přidána kolekce s časovými značkami pro danou instrukci. Tyhle instrukce pak prochází a nastavuje instrukce která se provedla. Jelikož list(z teorie grafů) je jediný, který obsahuje informace o pokrytí, tak se to prochází od listu směrem k uzlu.

Analyzátor nejprve zjistí, zda pro jednotlivé třídy existují data, které Jacoco sesbíral. To se provádí v metodě *createAnalyzingVisitor* ve třídě *Analyzer*. Metoda vytvoří instanci třídy *ClassAnalyzer*, která prochází strukturu třídy a pro metody vytvoří třídu *MethodAnalyzer*, která následně prochází strukturu metody. Při prochází a analyzování metody bylo potřeba, aby se dostala ke kolekci časových značek. Proto si musí referenci na kolekci předávat třídy tak, aby při analyzování byla přístupná. Po převedení instrukcí se následně prochází celá struktura(viz. předchozí odstavec) v metodě *visitEnd*. Zde se nejprve prochází ty instrukce, které obsahují informace o pokrytí. U nich se následně procházejí předchůdci a překopírují se data. Překopírovávání se provádí v metodě *setCovered*(viz. kód 20) ve třídě *Instruction*. Ta prochází jednotlivé předchůdce. Zde bylo přidat i kód, který překopíruje i časové značky. V metodě se pomocí metody *mergeTimestamps*, která spojí časové značky a následně je i protřídí od nejmenšího čísla po největší.

```
public void setCovered() {
   Instruction i = this;
   while (i != null && i.coveredBranches++ == 0) {
      final List<Long> list = i.getListOfTimestamps();
      i = i.predecessor;
      if (i != null) {
         if (IfProbes.isLineInTimestams(i.getLine()) && !IfProbes.isNotUsed(i.
             getLine())) {
            i.mergeTimestamps(IfProbes.getLineTimestamps(line));
         }
         i.mergeTimestamps(list);
      }
   }
}
```

Výpis 20: Procházení předchůdců instrukcí

Při pozdějším analyzování bylo zjištěno, že pokud podmínka *if* neobsahuje *else* blok, tak na řádku kde je podmínka, se objeví jenom ty časové značky, které jsou obsaženy v *then* bloku. Pro opravení byl využit princip na kterém funguje Jacoco v případě podmínky, který usnadní opravení. Jacoco invertuje podmínku(viz. kapitola 5.2.2), takže pokud podmínka neobsahuje *else* blok, tak Jacoco jej vytvoří. Zde je patrné, že *else* blok, bude jako *then* blok a tudíž, následující probe je z bloku *else*, který se využije pro kompletní časové značky. K tomu složí třída *IfProbes*. Jelikož celá analýza tříd se prochází postupně, tak pro jednoduchost je třída *IfProbes* statická, které se vždy na začátku označí, která třída se zrovna analyzuje. To se provádí v metodě *createAnalyzingVisitor*.

Pokud při analyzování se narazí na instrukci skoku, zavolá se metoda *visitJumpInsnWithProbe*. Zde se pak kontroluje zda instrukce je podmíněný skok(instrukce *IFxx*. Třída *IfProbes* obsahuje metodu(*isIfInsn*), která zkontroluje zda instrukce je typu *IFxx*. Metoda *visitJumpInsnWithProbe* obsahuje parametr *probeId*, který značí na jakém indexu se nachází následující probe. Tento parametr pak se využije pro získání časových značek, které se uloží k řádku, na kterém je daná instrukce.

Třída *MethodAnalyzer* si uchovává poslední zpracovanou instrukci v atributu *lastInsn*. Pokud analyzátor narazí na instrukci, kde je přidán probe, vloží do instrukce sesbíraná data v metodě *addProbe*.

Po převedení instrukcí na vlastní implementaci se procházejí všechny instrukce, které obsahují sesbíraná data a následně se procházejí jejich předchůdci. Při procházení předchůdců se jim překopírují nasbíraná data. To se provádí ve třídě *Instruction* v metodě *setCovered*. Pokud řádek instrukce je uložen ve třídě *IfProbes*, tak se k původním časovým značkám přidají i ty, které

jsou ve třídě *IfProbes*. To znamená, že instrukce je typu *IFxx* a je potřeba přidat k časových značkám další z *else* bloku. Kvůli tomu, že na jednom řádku může být více instrukcí, musí se kontrolovat, zda daná kolekce již nebyla použita.

Následně analyzátor prochází všechny instrukce. Zde se vezme řádek na kterém je daná instrukce a její časové značky. Tohle se uloží do hashovací tabulky. Kde klíčem je řádek a hodnota je časová značky. Jelikož na každém řádku je více instrukcí, tak se kontroluje, zda daný řádek je už v hashovací tabulce. Třída *MethodCoverageImpl* uchovává všechny analyzované data pro jednotlivé řádky. Proto se následně tabulka uloží do této třídy. Pro zjednodušení k přístupu časových značkám řádku, se následně časové značky vloží do hashovací tabulky celé třídy. To se provádí na konci „navštívení“, tj. metoda *visitEnd*. Zde se přidává pokrytí metody do pokrytí třídy a to metodou *addMethod*. V této metodě se následně zavolá *addTimestampsToClassLine*, která sloučí hashovací tabulky. Pro získání časových značek řádku pak stačí zavolat metodu *getLinesTimestamps* třídy, která implementuje rozhraní *IClassCoverage*.

5.3 Úvod do JDT a PDE

JDT, neboli Java Develepment Tools který je součástí Eclipse. JDT se rozděluje na několik komponent a každá z nich poskytuje jinou funkčnost nad celým Eclipse. Mezi to patří třeba spouštění programů, dále vytváření a manipulaci Java projektů nebo UI prvky Eclipsu(dialogy, view,..).

PDE, neboli Plug-in Development Environment poskytuje nástroj, pro vytváření, správu a spouštění modulů pro Eclipse.

5.3.1 Vytvoření kostry modulu

K vytvoření základní struktury modulu se vytvoří „Plug-in Project“ ve složce „Plug-in Development“ v Eclipse. Kde dále zadáme název modulu. Na další stránce se zadají identifikátory a verze modulu. A jako poslední můžeme využít již z existujících šablon. Tyto šablony pomáhají k pochopení jistých funkcionalit(jako například práce s tlačítky, s menu a nebo s view).

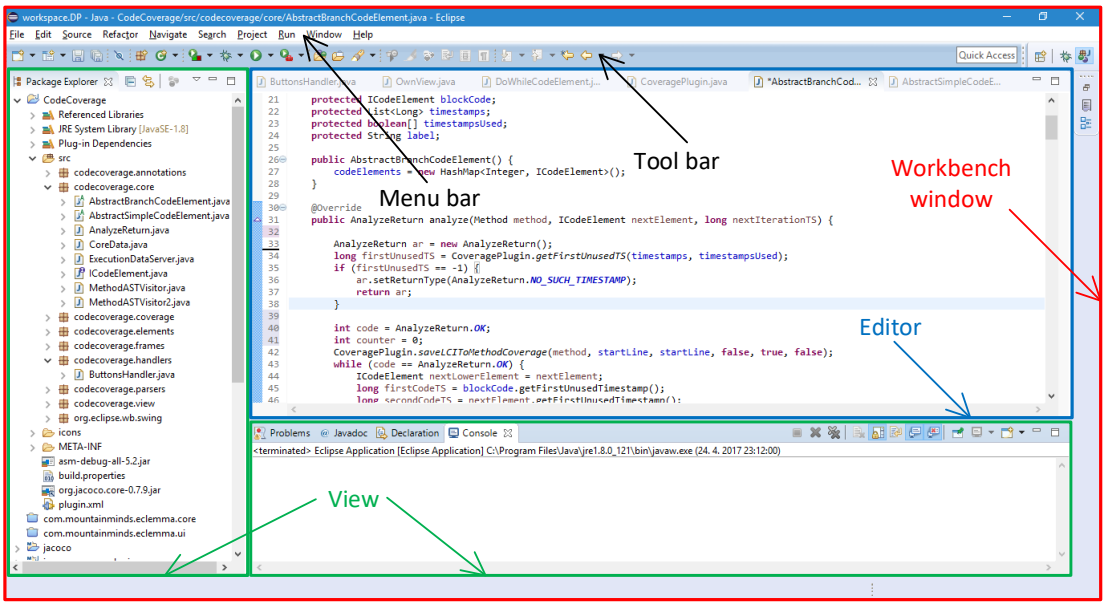
Moduly se dají i spustit přímo z Eclipse. Projekt se musí spustit jako „Eclipse Application“. Při prvním spuštění se vytvoří nový „workspace“ s výchozím jménem „runtime-EclipseApplication“ a spustí se nový Eclipse. Ten využívá zmíněný nový „workspace“. Ten se vytvoří v místě, kde je umístěn „workspace“ modulu. Pokud se nacházejí ve „workspace“ více modulů, všechny využívají stejný „workspace“ a spustí se nový Eclipse se všemi moduly. Pro vypnutí stačí projekt uzavřít(„closeProject“).

Nejdůležitějším konfiguračním souborem je „plugin.xml“. Ten obsahuje veškeré informace o modulu. Jako je například jaké knihovny využívá, tlačítka, pohledy(view) a podobně(viz. kapitola 5.6).

5.3.2 Popis struktury uživatelského prostředí

Nejdůležitější je třída *PlatformUI*. Ta zpřístupňuje přístup k uživatelskému rozhraní. Kvůli tomu, že získání aktivního editoru občas nefunguje pomocí třídy *PlatformUI*, využívá se třída *HandlerUtil*. Ta se využívá v metodách, které se spouští po stisknutí tlačítek v Eclipse, jelikož vyžaduje třídu *ExecutionEvent*.

- **Workbench window** - Celé okno prostředí
- **Editor** - Okno pro editace textu
- **View** - Pohledy, pro zobrazení různých dat
- **Menu bar** - Menu
- **Tool bar** - Panel nástrojů



Obrázek 24: Struktura odesílání časových značek

5.3.3 Popis třídy ProjectParser

Všechny metody, které získávají informace o projektu jsou uloženy ve třídě *ProjectParser* jako statické.

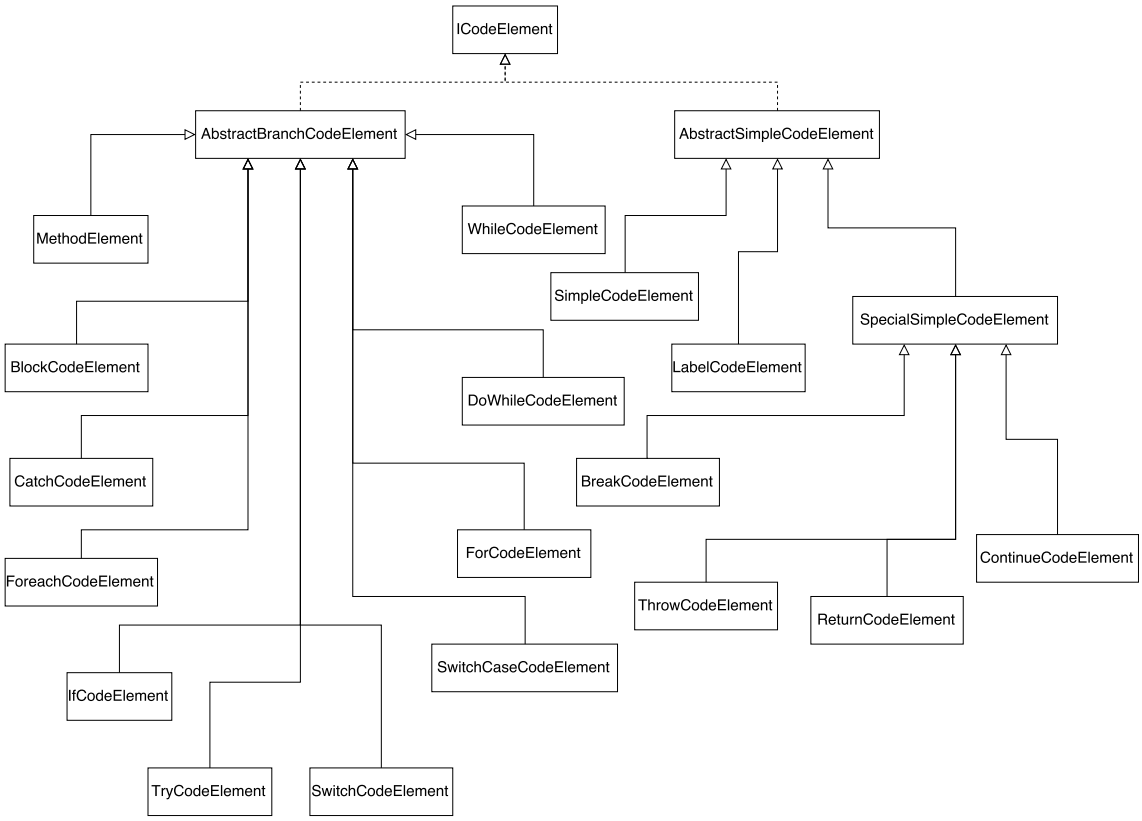
- **getPackages** - získání všech balíčků daného projektu
- **getCurrentWorkbenchPage** - získání aktivního WorkbenchWindow
- **getFileOfClass** - Získání souboru dané třídy
- **openEditorOfGivenProject** - Otevření editoru dané třídy v určitém projektu
- **openEditorOfCurrentProject** - Otevření editoru dané třídy v aktivním projektu
- **getClassPathSources** - Získání všech složek, ve kterém se nacházejí třídy projektu(tzv. „ClassPath“
- **isSourcePath** - Kontrola, zda je složka typu *ClassPath*
- **findOpenCurrentClass** - Získání třídy, která je otevřená
- **getCurrentFilePath** - Získání cesty k souboru, který je otevřený
- **getCurrentJavaProject** - Získání aktivního Java projektu

5.4 Analýza struktury kódu

Pro lepší vyhodnocení pokrytí cest kódu, se pro kód vytvoří stromová struktura kódu. K vytvoření vlastní stromové struktury se využívá API, které pracuje s AST(Abstract Syntax Tree). Je součástí JDT. AST je stromová struktura kódu, kde uzly jsou operátory a listy jsou operandy. Slovíčko abstraktní znamená, že stromová struktura kódu není úplně přesná. Například závorky ve stromové struktuře nemusí být, jelikož struktura stromu tuhle „funkci“ již zahrnuje. AST je podobné k DOM(Document Object Model)

Pro procházení struktury se využívá návrhový vzor Návštěvník(viz. obrázek 21 a kapitola 5.2.3). Pro procházení struktury kódu se využívá třída *ASTVisitor*. Kde pro každý operátor a operand existuje metoda *visit*. Všechny metody se jmenují *visit*, využívá se zde přetěžování metod(tj. určuje se podle parametrů metody). Například pro navštívení podmínky *if*, se zavolá metoda *visit(IfStatement node)*. Metody mají návratový typ *boolean*. Ten slouží k tomu, pokud nechme procházet potomky daného uzlu. Z parametru metody zle pak získat informace o uzlu(instrukci). Jako například kde začíná instrukce a jakou má velikost(textovou velikost), nebo uzel. Obsahuje dále i metodu *endVisit*. Ta se volá, pokud daný uzel se opouští(tj. celý jeho podstrom se prošel).

Pro vlastní stromovou strukturu byly vytvořeny třídy reprezentující instrukce. Pro jednoduchost se pro jeden řádek vytvoří třída, která jej reprezentuje. Třídy by se daly rozdělit na dvě skupiny. První skupina reprezentuje jednoduché instrukce(například vytvoření nové instance třídy, aritmetické operace, volání metod a podobně). Druhá skupina reprezentuje takové instrukce, které větví program(například *if*, *while*, *try* a podobně). Většina z tříd je „self-describing“. (viz. obrázek **??**)



Obrázek 25: Třídní diagram

- **SimpleCodeElement** - Představuje jednoduchou instrukci(vytvoření nové instance, aritmetické operace,..)
- **SpecialSimpleCodeElement** - Instrukce, které upravují průchod kódu, typ se určuje atributem *codeType*
- **MethodElement** - Kořen stromové struktury
- **AbstractBranchCodeElement** - Třída reprezentující větvení kódu
- **AbstractSimpleCodeElement** - jednoduché instrukce

K procházení třídy je potřeba kořenový uzel. Ten získáme ze třídy *IType*. (viz. kód 21

```
ICompilationUnit iCompilationUnit = type.getCompilationUnit();
CompilationUnit cu = parse(iCompilationUnit);
ASTNode rootNode = cu.getRoot();
```

Výpis 21: Získání uzlu z IType

Třída *IType* obsahuje metodu *accept*, která následně bude procházet celou strukturu a volat příslušně metody ve visitoru. K tomu je potřeba instance visitoru. Pro jednodušší analýzu struktury kódu, se nejprve získají uzly metod. K tomu slouží metoda *visit(MethodDeclaration)*. Jelikož chceme zjistit jenom tohle, vrátíme *false*. Tak aby se dál neprocházely jeho potomci. Ke každému uzlu se vytvoří třída *MethodElement*, která je uzlem nové struktury kódu. Dále má atributy pro začátek a konec řádku, název metody, název třídy a instanci na třídu *CompilationUnit*, která se využívá pro získání řádku. Následně se pak procházejí všechny metody a u nich se volá metoda *accept* s vlastní implementací *ASTVisitor*, *MethodASTVisitor*. *MethodASTVisitor* implementuje všechny metody *visit*, jelikož na řádku se může objevit jakákoliv instrukce.

Základním kamenem je zásobník, který si ukládá nynější uzel do kterého se vkládají další uzly, které se navštíví. Pokud daný uzel je již projitý, tak se uzel ze zásobníku odstraní pomocí metody *removeLast*. Pokud se narazí na uzle, který obsahuje další příkazy, uloží se do zásobníku. Jakmile se narazí na příkaz, tak se uloží do toho uzlu, který je na vrcholu zásobníku. Do zásobníku se vkládají ty třídy, které dědí ze třídy *AbstractBranchCodeElement*.

5.4.1 Jednoduché přikazy

Pro jednoduché přikazy, které neupravují průchod kódu, se zavolá metoda *visitCode*. Ta vytvoří novou instanci třídy *SimpleCodeElement*, jí nastaví začátek a konec řádku a vloží ji časové značky na daném řádku. Ze zásobníku získá vrchol zásobníku a do této třídy vloží nový příkaz. Zde se pak kontroluje, zda v daném uzlu už neexistuje nějaký příkaz na daném řádku. Je to kvůli tomu, že celý řádek se bere jako jeden příkaz. Kvůli tomuhle přístupu může dojít k chybné analýzy. Pokud programátor umístí na jeden řádek více příkazů, analyzátor to nepozná.

```
public void visitCode(ASTNode node) {
   AbstractBranchCodeElement el = stack.getLast();
   SimpleCodeElement code = new SimpleCodeElement();
   code.setStartLine(cu.getLineNumber(node.getStartPosition()));
   code.setEndLine(cu.getLineNumber(node.getStartPosition() + node.getLength())
       );
   code.setTimestamps(cc.getLinesTimestamps(code.startLine));
   el.addElement(code);
}
```

Výpis 22: Metoda visitCode pro jednoduché příkazy

5.4.2 Příkaz break

Příkaz break může obsahovat i návěstí. Proto se musí z parametru *BreakStatement* pomocí metody *getLabel* získat návěstí a pomocí metody *getIdentifier* se získá jméno návěstí. Ostatní příkazy jsou stejné jako u jednoduchých příkazů.

```
public boolean visit(BreakStatement node) {
   AbstractBranchCodeElement el = stack.getLast();
   String label = null;
   if(node.getLabel() != null) {
      label = node.getLabel().getIdentifier();
   }
   BreakCodeElement code = new BreakCodeElement(label);
   code.setStartLine(cu.getLineNumber(node.getStartPosition()));
   code.setEndLine(cu.getLineNumber(node.getStartPosition() + node.getLength())
       );
   code.setTimestamps(cc.getLinesTimestamps(code.startLine));
   el.addElement(code);
   return super.visit(node);
}
```

Výpis 23: Metoda visit pro příkaz break

5.4.3 Návěští

Při průchodu návěštím se zavolá metoda *visit(LabeledStatement node)*. Zde získáme jméno návěští. Kvůli tomu, že dál se navštíví další metoda *visit* a musí se zajistit, že daná metoda nepřidá příkaz na daný řádek. To se zajišťuje atributem *labeledVisit*, který se nastaví na *true*. Při analýze, jak se prochází návěští, bylo zjištěno, že další metoda která se navštíví je *visit(SimpleName node)*, proto se zde kontroluje zda atribut *labeledVisit* není nastaven na *true*. V metodě *endVisit(SimpleName node)* se pak nastaví atribut na *false*.

5.4.4 Ostatní přikazy

Pro instrukce *return*, *continue* a *throw* je stejný kód jako pro jednoduché příkazy(viz. kapitola 5.4.1). Nevyžadují pro inicializaci speciální nastavení.

5.4.5 Blok

Blok se řadí mezi skupinu, která větví program. Je to proto, že obsahuje další příkazy. Pomocí metody *addElement* se přidá nový příkaz do bloku. Ten se ulož do listu. Zde se zjišťuje, zda na daném řádku už neexistuje příkaz. Pokud na daném řádku už existuje příkaz, metoda vrátí *false*.

```
public boolean visit(Block node) {
   BlockCodeElement code = new BlockCodeElement();
   code.setStartLine(cu.getLineNumber(node.getStartPosition()));
   code.setEndLine(cu.getLineNumber(node.getStartPosition() + node.getLength())
       );
   code.setThisNode(node);
   code.setTimestamps(cc.getLinesTimestamps(code.startLine));
   AbstractBranchCodeElement element = stack.getLast();
   if(element.addElement(code)) {
      stack.add(code);
   }
   return super.visit(node);
}
```

Výpis 24: Metoda visit pro příkaz break

Při skončení procházení kódu(metoda *endVisit*)(viz. kód 25) v tomhle bloku se zjišťuje, zda daný blok je blok celé metody. Jelikož pokud metoda nemá návratový typ, tak na posledním řádku metody(na řádku kde je ukončovací znak pro blok) se nachází ještě časové značky pro opouštění metody. To se zjišťuje tak, že pokud na zásobníku jsou dva prvky, tak se jedná o blok metody. Následně se vytvoří nový příkaz(*SimpleCodeElement*). A vloží se do listu příkazů v bloku.

```
public void endVisit(Block node) {
   BlockCodeElement code = (BlockCodeElement) stack.getLast();
   if(stack.size() == 2) {
      code.setBlockWithEndTimestamp(true);
   }
   if(code.isBlockWithEndTimestamp()) {
      SimpleCodeElement element = new SimpleCodeElement();
      element.setStartLine(code.endLine);
      element.setEndLine(code.endLine);
      element.setThisNode(null);
      element.setTimestamps(cc.getLinesTimestamps(element.startLine));
      code.addElement(element);
   }
   stack.removeLast();
   super.endVisit(node);
}
```

Výpis 25: Metoda visit pro příkaz break

5.4.6 Cykly

Všechny cykly jsou stejné z pohledu struktury a nepotřebují některé speciální nastavení. Nastaví se jím návěští, které je uloženo v atributu *currentLabel*. (viz. kód 26)

```
@Override
public boolean visit(ForStatement node) {
   ForCodeElement code = new ForCodeElement();
   code.setStartLine(cu.getLineNumber(node.getStartPosition()));
   code.setEndLine(cu.getLineNumber(node.getStartPosition() + node.getLength())
       );
   code.setThisNode(node);
   code.setTimestamps(cc.getLinesTimestamps(code.startLine));
   code.setLabel(currentLabel);
   currentLabel = null;
   AbstractBranchCodeElement element = stack.getLast();
   if(element.addElement(code)) {
      stack.add(code);
   }
   return super.visit(node);
}
```

Výpis 26: Metoda visit pro příkaz break

5.4.7 Odchytávání výjimek

Kód pro příkaz *try* je stejný jako pro cykly. Zde se liší metodou *addElement*. Jelikož *baf* obsahuje jak blok pro kód, tak i bloky *baf*. V metodě se kontroluje zda element je instance typu *blockCodeElement*, pokud ano, tak se dále kontroluje zda, už *try* obsahuje blok pro kód. Pokud ne, vloží se do proměnné *blockCode* a pokud ano, vloží se do atributu *finallyBlock*. Z názvu jde poznat, že jde o *finally* blok. Pokud není instance typu *blockCodeElement*, tak se kontroluje na typ *CatchCodeElement*. Pokud ano, vloží se do atributu *codeElements*. Jiném případě vrátí metoda *false*.

5.4.8 Konstrukce switch

Kód pro vložení klíčového slova *switch* je stejný jako u cyklů. Zde se liší v kódu pro ukládání jednotlivých možností. Zde je problém v tom, že celý kód pro danou možnost(*case*) není uložen v uzlu *SwitchCase*. Proto se musel najít způsob jak to obejít. Při volání metody *visit(SwitchCase node)* se kontroluje, zda na zásobníku je už třída *SwitchCaseCodeElement*. Pokud ano, ta se ze zásobníku odstraní. Tohle indukuje, že analyzátor přešel k další možnosti. Tohle se dál kontroluje po opouštění celého *switch*.

Dále je pak potřeba opravit poslední možnost(*case*), protože na posledním řádku(příkaz *break*) není uloženy časové značky. Proto se zkopírují z předposledního kroku. Zde je patrné, že tohle může způsobit ve špatnou analýzu časových značek. Oprava se provede zavoláním metody *repairLastCase*.

5.5 Vyhodnocení pokrytí cest

Po tom, co se vytvoří stromová struktura se následně pomocí metody *analyze* začne analyzovat časové značky. Metodu implementují všechny třídy reprezentující příkazy/větvení. Metoda obsahuje tři parametry. První je reference na třídu *Method*. Do ní se vkládají informace jak daná metoda se vykonala. Tahle metoda se dále využívá pro zobrazení stromové struktury ve view(viz. kapitola 5.6.7). Druhý parametr je reference na příkaz, který následuje v předešlém bloku a třetí parametr je další časová značka, kdy začíná další průběh metody. Ten se využívá k lepší detekci odchytávání výjimek(viz. kapitola 5.5.7). Jeho návratový typ je *AnalyzeReturn*. Do ní se vkládají informace o tom, jak se daný příkaz/blok provedl. Zda se provedl v pořádku, provedl se příkaz pro změnu chodu kódu nebo se vyskytla výjimka. Typ kódu se určuje pomocí atributu *returnType*. Pro jednoduché čtení typu se na místo číselného kódu využívají statické proměnné. Typy kódu:

- **RETURN** - Analyzátor narazil na příkaz *return*
- **BREAK** - Analyzátor narazil na příkaz *break*
- **CONTINUE** - Analyzátor narazil na příkaz *continue*
- **THROW** - Analyzátor narazil na příkaz *throw*
- **NOT_THIS_BLOCK** - Pokud daný blok se neprovedl a má se pokračovat v analýze dál
- **NO_SUCH_TIMESTAMP** - Příkaz/blok neobsahuje další časové značky
- **OK** - Příkaz/blok se provedl v pořádku
- **THROWS** - Při analýze byla zjištěna výjimka

Tříd *AnalyzeReturn* obsahuje dále ještě atribut *breakLabel*, který se využívá v případě příkazu *break*, pokud odkazuje na nějaké návěští a dále obsahuje atribut *throwTS*, který obsahuje časovou značku posledního příkazu na kterém došlo k výjimce. Ten se využívá k zjištění, zda výjimka se odchytla a nebo se postoupí dál.

Celý princip funguje na tom, že se procházejí jednotlivé příkazy jak jdou za sebou a porovnávají se časové značky. Pokud daný příkaz se provedl, vloží se jeho číslo řádku do listu. Pokud se zjistí, že daný příkaz se provedl, tak se označí časová značka jako použita. K tomu

složí pole(atribut *timestampsUsed*) typu *boolean*, kde index je stejný jako u časové značky. Na daný index se uloží *true*.

Pro další dodatečné informace o průběhu kódu se uchovává pro každý řádek třída *LineCoverageInfo*. Ta si uchovává začátek a konec příkazu, dále zda se jedná o třídu dědící ze třídy *AbstractBranchCodeElement* a kolikrát se daný řádek provedl celkově.

Metody které se využívají v průběhu analýzy:

- *getFirstUnusedTimestamp* - Získá první nevyužitou časovou značku, jinak *-1*
- *getSecondUnusedTimestamp* - Získá druhou nevyužitou časovou značku, jinak *-1*
- *saveLCIToMethodCoverage* - Uloží informace o pokrytí(viz. předešlý odstavec) a pokud se jedná o příkaz, kde se vyhodila výjimka, tak se řádek vynásobí *-1*
- *searchAndTagReduntTS* - vyhledá stejné časové značky a označí je jako použité

5.5.1 Třída MethodElement

Metoda *analyze* ve třídě *MethodElement* je velmi jednoduchá. Zde se opakovaně volá na blok metoda *analyze*, pokud nevrátí kód, že blok neobsahuje již další časové značky. Dále se zde pro každý průchod metodou vytvoří nový list do kterého se ukládají řádky, které se prošly. To se provede zavoláním metody *incCountOfCoverages*. Pro začátek metody se vždy uloží číslo -1 do listu. Pro správnou analýzu výjimek se zjistí druhá časová značky prvního příkazu v metodě. To se provede metodou *getSecondUnusedTimestamp* na blok metody.

Po celkovém zjištění průchodu metody se zavolá metoda *analyzeListOfCoverage*. Ta upraví a odstraní ten list, která obsahuje jenom jeden prvek. Ten označuje, že se jedná o poslední průchod analýzy metody, kde bylo zjištěno, že neobsahuje již další nepoužité časové značky a sníží celkový počet průchodů metody. Dále projde každý průchod a místo *-1* vloží řádek začátku metody a informaci o ní (třída *LineCoverageInfo*)(viz. předešlá kapitola).

5.5.2 Blok

V bloku se projdou všechny příkazy. V první části se vytáhnou tří elementy. První je ten, který se zrovna analyzuje a dva další se ho následují. To se provede statickou metodou *getNextCode*, která dle dané kolekce elementu, začátek hledání a konec metody. Dva poslední se následně kontrolují, ze kterého se má využít časová značka, která se pak předá jako parametr do analýzy. To se provádí kvůli tomu, že pokud jako další element je třída dědící z *AbstractBranchCodeElement*. Pokud by se daný blok přeskočil, tak do analýzy by se předala špatná časová značka. Jako poslední před zavolání metody *analyze* se zjišťuje, zda daný element není poslední. Pokud ano, tak po analýze se ukončuje analýza daného bloku. Zde se kontroluje jestli se příkaz provedl správně. Pokud ne, tak jeho chyba se předá výš(vrací se jeho kód).

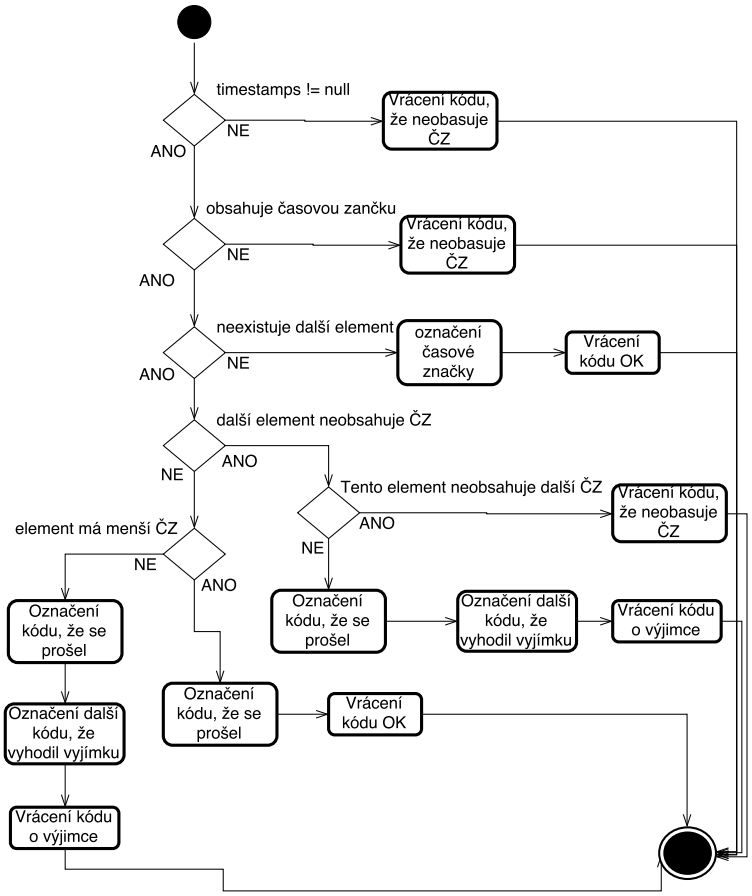
5.5.3 Jednoduchý příkaz

Metoda *analyze* se zde nepřekrývá a využívá se ze třídy *AbstractSimpleCodeElement*.

První kontrola je, zda element obsahuje nějakou volnou časovou značku a pokud ne, vrátí kód *NO_SUCH_TIMESTAMP*. Pokud další element neexistuje je zřejmé, že tento element je poslední. Proto se následně označí poslední časová značka a vrátí se kód *OK*.

Pokud další element existuje a neexistuje v něm další volná časová značka, tak můžou nastat dvě situace. Pokud ani tento element nemá volnou časovou značku, tak se vrátí kód *NO_SUCH_TIMESTAMP*. Tohle může nastat jenom v případě, že se element nachází jako první v metodě a další průchod metodou neexistuje. Druhá varianta je ta, že na tomto elementu existuje časová značka. To ale značí, že další příkaz se neprovedl a proto se musela vyhodit výjimka na dalším příkazu. Takže se označí časová značka jako použitá a uloží se data o tomto řádku a o dalším řádku. Ten se označí, že zde byla vyhozena výjimka(jako poslední parametr v metodě *saveLCIToMethodCoverage*).

Poslední varianta je ta, že oba příkazy obsahují časové značky. Zde se musí porovnat, zda následující příkaz neobsahuje první nepoužitou časovou značku, která ale se využije v dalším průchodu metodou. To by znamenalo, že na tomto příkazu se vyhodí výjimka. Toto porovnání se provede tím, že se vytáhne druhá nepoužitá časová značka u tohoto příkazu a porovná se, zda není menší než v druhém příkazu. Pokud ano, označí se metoda za provedou a další příkaz, že zde byla vyhozena výjimka. Pokud ne, tak příkaz se provede v pořádku, označí se a pokračuje se dál.



Obrázek 26: Třídní diagram

5.5.4 Příkazy ovlivňující chod programu

Zde patří příkazy *return*, *break*, *continue* a *throw*. Zde analýza je velmi jednoduchá. Zkontroluje se, zda příkaz se provedl(zda obsahuje nepoužitý časovou značku). Označí se a do třídy *AnalyzeReturn* se vloží kód daného příkazu z atributu *codeType*. Příkaz *break* se liší jenom v tom, že se uloží i návěstí, pokud obsahuje.

5.5.5 Cykly

Všechny cykly, kromě typu „do-while“, se v ničím neliší a jejich metoda *analyze* je stejná. Zde je analýza velmi složitá. Může zde nastat hodně situací. Cyklus neobsahuje jednotlivé elementy(příkazy), ale celý blok(*BlockCodeElement*) a nebo jeden příkaz. Je to kvůli rozdělení kódu.

Pro usnadnění se nacházejí i časové značky na řádku, kde se kontroluje zda se má vejít do cyklu. Ulehčuje to získání časových značek pro předešlé příkazy/bloky. Pokud se cyklus

provedl správně(i při nevykonání bloku cyklu), označí se časová značka jako použitá metodou *setFirstUnsedTSUsed*.

Nejprve se testuje, zda se provedla kontrola podmínky. Pokud ne, tak se vrátí kód *NO_SUCH_TIMESTA*. Následně se inicializuje počítadlo kolikrát se provedl cyklus. Jelikož se ví, že se testovala podmínka, tak se daný řádek označí a přidají se k němu informace. A následně se celý blok prochází dokud se nezjistí, že byl proveden speciální příkaz k ukončení nebo nějak upravení cyklu a nebo cyklus se ukončil a má se pokračovat dále.

Jelikož metoda *analyze* potřebuje další element, musí se nejprve testovat, zda cyklus se bude opakovat. To se provede jednoduchou podmínkou. Testuje se zda druhá nepoužitá časová značka je menší než časová značka následujícího příkazu/bloku. Zde je samozřejmě nutné kontrolovat, zda druhá časová značka existuje(není rovna *-1*). Pokud se cyklus bude opakovat, vytvoří se umělý element, který obsahuje druhou nepoužitou časovou značku. Tento element se následně předává analýze jako další příkaz po dokončení bloku.

Jako první situace může nastat, že první kód v bloku a příkaz nacházející se za blok neobsahují další volnou časovou značku. To znamená, že se při vykonání podmínky vyhodila výjimka.

Druhá situace může nastat, že první kód obsahuje nějakou novou časovou značku a tudíž se může blok analyzovat. Jelikož může se narazit na příkazy *break*, *continue* nebo *return* které by ovlivnili průchod cyklem, musí se testovat jaký kód vrací blok. Pokud narazí na *return*, kód se pošle dál. Jestliže narazí na *break* a nebo *continue*, tak v prvním případně, se testuje, zda neobsahuje návěští(jestli vůbec obsahuje), které patří tomuto cyklu a cyklus se opustí. V případ *continue* se provádí blok stejným způsobem.

Další situace je opakem předešlé situace. Pokud blok neobsahuje žádnou volnou časovou značku, tak se blok dál neprovádí a může se opustit celý uzel. Pokud se prošel alespoň jednou, tak se vloží informace o dalším průchodu cyklu a inkrementuje se atribut, který značí, kolikrát se daný cyklus prošel. Testování zda se cyklus prošel alespoň jednou je proto, že hned na začátku se vloží informace, že se vykonala podmínka cyklu. Kdyby se to nekontrolovalo, redundantně by se vložila informace o podmínce a zvýšil by se počet cyklů.

Při analyzování bylo zjištěno, že analyzátor vloží časové značky, které tam nemají co dělat a jsou stejné. Proto se zavolá na blok metoda *searchAndTagReduntTS* a ta odstraní špatně vložené časové značky a cyklus se opustí a označí se, že cyklus(uzel) proběhl v pořádku.

Jako předposlední situace může nastat, že oba elementy obsahují nevyužité časové značky. Zde se musí kontrolovat, zda se má vejít do cyklu a nebo z něj odejít. V prvním případně se případně uloží informace o dalším průchodu cyklu a analyzuje se blok. Tento krok je stejný jako druhá situace (třetí odstavec nad tímto). V druhém případně to znamená, že časová značka mimo cyklus a proto se má odejít z cyklu. Takže se označí případně další cyklus a označí se časová značka u cyklu, že byla použita.

Algoritmus se dostane za cyklus *while* jenom v případě, že vše proběhlo v pořádku. Proto se vrátí kód *OK*.

5.5.6 Do-while

Cyklus typu „do-while“ je mnohem jednodušší v porovnání s ostatními cykly. Jelikož se musí blok alespoň jednou provést, situací je to mnohem miň.

Nejprve se analyzuje blok cyklu. Následně se testuje jaký kód blok vrátil. Pro *return* se hned opustí cyklu. Pro *break* se opustí cyklu a pokud obsahuje návěští pro tento cyklus, tak se opustí a vrátí kód *OK*. Pokud ne, vrátí celou instanci třídy co vrátil blok. Až následně se testují časové značky zda se má daný cyklus provést znova. Pokud obsahuje časovou značku, která je menší než časová značka elementu za cyklem, tak se provede cyklus provede znova. Pokud ne, tak se do proměnné *code* vloží kód, aby se cyklus *while* opustil. A jako poslední se uloží informace o průchodu cyklu(metoda *saveLCIToMethodCoverage*).

Pokud se cyklus *while* opustí, tak se vrátí kód *OK*. Zde se algoritmus dostane jenom v případě, že vše proběhlo v pořádku.

5.5.7 Blok try

Analýza bloku *try* je velmi jednoduchá. Nejtěžší je poznat do kterého(a jestli vůbec) bloku *catch* skočí výjimka.

Jako první se blok analyzuje. Pokud se vrátil kód *OK*, tak se blok(uzel) ukončí.

Pokud ne a analýza vrátí kód *THROW* nebo *THROWS*, musí se nejprve poznat, zda výjimka se má odchytit a nebo postupovat dále. K tomu slouží parametr *nextIterationTS*, který značí časovou značku dalšího průchodu metody. Pomocí ním se dá poznat, zda se výjimka odchytí. To se pozná tak, že v některém bloku *catch* je časová značka menší než parametr. To se provede metodou *getCatchBlock*, které předáme časovou značku instrukce, která byla jako poslední a časovou značku další iterace metody. Ta prochází všechny bloky *catch* a najde, zda se má provést blok *catch*. V jiném případě se vrátí hodnota *null*. Pokud se má výjimka odchytit, tak se daný blok analyzuje. Dále pokud konstrukce *try* obsahuje blok *finally*, tak i ten se analyzuje.

Pokud se objeví výjimka buď v *catch* bloku nebo *finally* bloku, tak se musí daný kód předat dál. Pokud se vyskytne výjimka v obou blocích, tak přednost má blok *finally*. V jiném případně se vrací kód z bloku *catch* a pokud výjimka se má předat dál, tak se pošle kód z bloku *catch*.

Jelikož i konstrukce *try* může obsahovat návěští, musí se zkontrolovat, zda návěští patří k této konstrukce. Pokud ne, kód se pošle dál.

Jakýkoliv jiný kód se posílá dál.

5.5.8 Konstrukce switch

Při analýze konstrukce *switch* se vyhledá nejprve ten blok, který má nejnižší nevyužitou časovou značku. Ta se následně porovná, zda je menší než časová značka následujícího elementu(příkaz/blok).

Nejprve se testuje zda konstrukce má nějaký nevyužitou časovou značku. Pokud ne, tak to indikuje, že konstrukce je první a tak se vrátí kód *NO_SUCH_TIMESTAMP*. Pokud má, tak se konstrukce určitě provede(pokusí se najít blok pro daný parametr), takže se označí řádek(kde

se nachází začátek konstrukce), že byl vykonán a přidán řádek do pole cesty metodou. Dále se kontroluje zda konstrukce obsahuje alespoň jeden blok *case*. Jelikož kolekce *Set* neumožňuje indexaci, musí se pomocí iterace vzít první prvek. Následně probíhá algoritmus, který vyhledá blok s nejnižší časovou značkou. Musí se samozřejmě kontrolovat, zda daný blok obsahuje nějakou nevyužitou časovou značku. Zde můžou nastat tři situace.

První situace je ta, že žádný blok neobsahuje nevyužitou časovou značku a proto se označí konstrukce, že se provedla(její první řádek) a ukončí se analýza tohoto elementu.

Druhá situace je, že našel se blok s nejnižší časovou značku, ale ta je větší než ta u dalšího elementu(příkazu/bloku). To znamená, že kód nevešel do konstrukce. Takže se označí konstrukce že se provedla a ukončí se analýza.

Poslední situace je ta, že se našel blok, který se má analyzovat(má nejnižší časovou značku a je menší než ta u dalšího elementu). Po analyzování bloku se vrátí kód, který vrátil blok a označí se konstrukce, že se provedla.

5.5.9 Vypočítání informací o pokrytí

Po skončení analýzy metody se vypočítají maximální počet cest metody a kolik ve skutečnosti se cest provedlo.

V prvním případě se zavolá metoda *getCountBranches* na metodu. Ta prochází celou strukturu metody a vypočítají se možnosti, jak se daný element může projít.

Jednoduchý příkaz vrací jedničku. Jelikož má jenom jednu možnost jak ho projít. U cyklů se sečte možnosti které vrátil blok a jednička. Jelikož cyklus se může přeskočit. U konstrukce *switch* se sečtou jednotlivé možnosti bloků *case* plus jedna. Jelikož se může *switch* přeskočit. U bloku se jednotlivé možnosti elementů násobí. U konstrukce *try* se vrátí možnosti bloku.(teorie viz. kapitola 3.2)

V druhém případě by se mohlo vzít počet průchodů metodou. Zde se musí dát pozor na ty průchody, které jsou stejné. Jelikož se jedná o ten samý průchod, musí se započítat jenom jednou. To se zjistí pomocí metody *calcCountOfPathCoverage*, která zkontroluje duplicitní průchody metody.

Metoda funguje na jednoduchém principu. Vytvoří se pole, které určuje, zda daný průchod je unikátní(nastaví se *true*). Následně se porovnávají jednotlivé průchody. Pokud se narazí na stejný průchod, přičte se jednička a druhý průchod se označí, že není unikátní. Tento průchod se pak přeskočí, jelikož se ví, že už je započítaný.

5.6 Modul pro Eclipse

Součástí diplomové práce je i modul do Eclipse, který dokáže spustit a analyzovat *JUnit* testování. Jeho vytvoření a základní informace o JDT a PDE viz. kapitola 5.3.

5.6.1 Import potřebných knihoven

Ke spuštění Jacoco v modulu, je potřeba zadat cestu k potřebných knihovnám. To se prování v souboru „plugin.xml“ v záložce „Runtime“. V záložce se nachází pro import knihoven do „classpath“ pod názvem „Classpath“. Zde se pomocí tlačítka „Add..“ vyberou potřebné knihovny, které se nacházejí v projektu modulu. Knihovny se mohou zde zkopírovat, ale je taky možné zde nahrát referenci na dané soubory. Takže při novém zkompilování knihovny se nemusí znova zkopírovat knihovna. Dále pod názvem „Exported Package“ ve stejné záložce se vyberou ty balíčky, které chceme přidat s modulem. K modulu se přidá jak Jacoco, tak se musí stáhnou ti knihovna s názvem „ASM“.

5.6.2 Vytvoření tlačítek v Tool bar

Definování tlačítek se provádí v souboru „plugin.xml“. Jsou dva způsoby. První způsob je přímo editovat xml soubor a nebo v záložce „Extensions“.

K definování a funkčnosti tlačítek je potřeba definovat více věcí. Prvně se musí vytvořit „toolbar“(identifikátor je *my.Toolbar*), do kterého se vloží tlačítka. Ten se vloží do „org.eclipse.ui.menus/toolb

Dále se definuje tlačítko pro tento „toolbar“. To se definuje v cestě „org.eclipse.ui.menus/toolbar:my.Toolb Kde první část je kde tlačítko bude(v tomto případně v „toolbar“) a za dvojtečkou je identifikátor. Zde se vytvoří *command*. Nejprve se mu nastaví identifikátor tlačítka(*id*), identifikátor příkazu(*commandId*), ikona tlačítka(*icon*), nápověda tlačítka(*tooltip*) a jako poslední je styl(*style*) tlačítka. V tomto případně „pulldown“. To znamená, že tlačítko se půjde rozkliknout.

Pro menu k tlačítku v „toolbar“ se zadají tlačítka zvlášť a to v cestě „menu:my.Toolbar.command1?after= Kde první část znamená, že se jedná o menu, další část je identifikátor „toolbar“, za ním je identifikátor příkazu. Zde je stejné nastavení jako předtím. Zde musíme nastavit text pomocí vlastnosti *label*.

Jako poslední je potřeba vytvořit tzv. „handler“. Je to třída, která se zavolá po stisku tlačítku. To se definuje v „org.eclipse.ui.handlers“. Kde v každém se definuje pro jaký příkaz(*commandId*) se to má provést a jaká třída(*class*) se o tyto příkazy stará.

Pro potřeby diplomové práce se vytvoří tři tlačítka. První tlačítko spustí program a provede analýzu. Druhé tlačítko je pro spuštění předešlého programu a třetí je spuštění programu, ale neprovede se analýza. Do konzole se vypíšou jednotlivé data, které se nasbíraly.

Všechny tlačítka obslouží třída *ButtonHandler*. Třída musí dědit ze třídy *AbstractHandler* a musí obsahovat metodu *public Object execute(ExecutionEvent event)*. Pomocí následujících metod „event.getCommand().getId()“, lze získat identifikátor příkazu, který jej vyvolal.

V prvním případě tlačítka se zavolá metoda *runJUnitTest*, které se předá parametr *event*, zda se má spustit předešlý program a zda se mají analyzovat data a nebo vytisknout data.

V druhém případě se zavolá ta stejná metoda, ale jako poslední parametr se vloží *true*, jelikož chceme jenom vytisknout nasbíraná data.

V třetím případě se zavolá ta stejná metoda, ale jako druhý parametr se vloží *true*, jelikož chceme spustit předešlý program. V metodě se kontroluje, zda byl už spuštěn nějaký program.

5.6.3 Server pro sběr dat

Pro sběr dat od agenta se využívají kódy, které jsou obsaženy v archivu při stažení Jacoco. Ty se nacházejí ve složce „doc/example/java“. Zde se využívá server, který čeká na připojení od agenta a následně příjme data. Ten je naimplementovaný ve třídě *ExecutionDataServer*. V této třídě jsou dva důležité atributy. První nastavuje cestu k souboru, kde se mají data uložit a druhý na jakém portu má server poslouchat. Po skončení sběru se data nacházejí v třídní proměnné *store*.

5.6.4 Spuštění JUnit testování

Při každém spuštění metody *runJUnitTest* se resetují(vymažou) všechny data v třídě *IfProbes*.

Nejprve je získán aktivní projekt. Ten se získá metodu *getCurrentJavaProject* ve třídě *ProjectParser*. Pro následné analyzování je třeba získat všechny balíčky v v projektu. To se provede metodou *getPackages*. Pro spuštění daného souboru(třídy) se musí získat třída, která je otevřená(aktivní okno editoru). To se získá metodou *findOpenCurrentClass* v daném projektu. Pokud danou třídu nenašel, v proměnné *classType* je *null* a zobrazí se dialog pomocí metody *showDialog*, které předáme jaký text se má zobrazit.

Následně spustíme server, který po skončení programu příjme nasbíraná data. K tomu se využívá třída *ExecutionDataServer* na které se zavolá metoda *run*.

Samotné spuštění je velmi složité. Proto se našel na internetu([6]) návody, jak spustit na dané třídě JUnit. Nejprve se zavolá metoda *createLaunchConfiguration*, která nastaví konfiguraci co se má spustit a jak. Metoda se musela upravit, jelikož chceme při spuštění přidat i agent(viz. kapitola 5.1). To se provede kódem 27 .

```
wc.setAttribute(IJavaLaunchConfigurationConstants.ATTR_VM_ARGUMENTS,
          "-javaagent:d:/jacocoagent.jar=output=tcpclient,address=localhost,
             port=9999");
```

Výpis 27: Vložení JVM argumentu

Po vytvoření konfigurace(třídy *ILaunchConfigurationWorkingCopy*) se spustí metodou *launch*, které předáme parametr, který určuje zda se má spustit v módu „debug“ a nebo normálně a druhý parametr referenci na *IProgressMonitor*. Jelikož žádný nepoužíváme, vloží se *null*. Aby po ukončení programu se vykonalo buď analýza a nebo zobrazení výsledků v konzoli, tak se vytvoří metoda, která reaguje na určité události. To se provede zaregistrování nové třídy *IDebugEventFilter*, ve které implementuje metodu *filterDebugEvents*. Metodě zjišťujeme o jakou událost se jedná. V tomhle případě chceme po ukončení programu(*DebugEvent.TERMINATE*). Následně pokud se mají jenom vypsat data na konzoli zavolají se metody *runAnalysisData*, která vloží

data do kolekce a *printExeData*, která vypíše jaké řádky se vykonaly a další informace z analyzátoru. Tohle slouží spíš pro kontrolu a používalo se to při vývoji. Pokud se má analyzovat pokrytí cest, tak se zavolá metoda *analyzeMethods*. Jako poslední se pak uloží cesta k souboru pro případné zavolání předchozího JUnit programu.

5.6.5 Spuštění analýzy dat

Analýza pokrytí cest se provádí v metodě *analyzeMethods*. Zde se nejprve provede Jacoco analýza, kde převede jednotlivé probe a časové značky na jednotlivé řádky(viz. kapitola 5.1). Analyzují se všechny třídy v projektu. Dále se označí třídě *IfProbes*, že se nejedná o offline mód a resetují se mu kolekce, která uchovává, které časové značky se použily.

Následně se procházejí všechny třídy, které se analyzovaly. Pro každou třídu se najde referenci na třídu(třída *IType*), která se využije pro analyzování struktury metody. Pro nalezení třídy se využívá metoda *findType* v daném projektu(třída *IJavaProject*).

Jako další se procházejí všechny třídy a ukládají se metody v dané třídě pro analýzu. Pro získání kořene celé třídy pro analýzu se musí nejprve daná třída analyzovat a vytvořit AST. Nejprve se z třídy získá *ICompilationUnit*. To se následně předá metodě *parse*, která vrátí *CompilationUnit*.

K parsování se využívá třída *ASTParser*. Ta se vytvoří pomocí statické metody *newParser* ve třídě *ASTParser*. Jako parametr se jí předá v jaké verzi Javy je kód. Ta vrátí instanci na *ASTParser*. Té předáme o jaký typ se jedná, parsovanou třídu a nastavení bindování. Následně pomocí metody *createAST* se vytvoří *CompilationUnit* pomocí kterého získáme kořen AST.

Kořen získáme pomocí metody *getRoot*. Vytvoří se instance třídy *ASTVisitor* a zavoláme metodu *accept* (viz. kapitola 5.4 a 5.4.8). Veškerá data se ukládají do tříd *Project*, *Class* a *Method*. Ty se využívají jak pro uložení informací o pokrytí cest tak i pro vykreslení stromové struktury v view(viz. kapitola 5.6.7).

5.6.6 Výpis nasbíraných dat

Pro kontrolu a výpis nasbíraných dat se využívá metoda *printExeData*. Metoda nejprve projde kolekci všech nasbíraných probe a vytiskne je do konzole.

Druhá část provede analýzu co Jacoco poskytuje. (viz. kapitola 5.1)

5.6.7 Vytvoření view

Pro výpis všech analyzovaných dat se využívá „view“ ve kterém se vytvoří stromová struktura všech projektů, které se analyzovaly. Zde se pak dostaneme k výsledkům analýzy.

Definování view se vytváří v souboru „plugin.xml“. Je to stejné jako u tlačítek. Definování view je velmi jednoduché, stačí přidat jednoduchý xml kód. Ten se nachází v „org.eclipse.ui.views“, ve kterém se nacházejí definice view.

```
<extension
   point="org.eclipse.ui.views">
    <view id="codecoverage.view.coverageview"
         name="Coverage view"
      class="codecoverage.view.OwnView"
      icon="icons\view.gif"/>
</extension>
```

Výpis 28: Definování view v plugin.xml

- **id** - Identifikátor view
- **name** - Název view
- **class** - Třída která spravuje view
- **icon** - Ikona view

Třída *OwnView*, která dědí z *ViewPart* se stará o vykreslení obsahu do view. To se provádí v metodě *createPartControl*.

Pro zobrazení stromové struktury projektů se využívá *TreeViewer*. Dále se využívá třída *Tree*, která se nastaví jako potomek *TreeViewer*. Pomocí metody *setHeaderVisible* se nastaví, aby se zobrazovaly popisky sloupců. Dále se vytvoří sloupec pomocí třídy *TreeColumn*. Té se nastaví její šířka, na jakou stranu se má text zarovnat, viditelnost řádků a název sloupce.

To jak se má vykreslovat stromová struktura se provádí pomocí dvou tříd. První je *ProjectContentProvider*. Ta určuje strukturu stromu. Druhá třída je *TableLabelProvider* a ta určuje, jaký text se má vypsat.

První třída musí implementovat rozhraní *ITreeContentProvider*. Zde jsou nejdůležitější tři metody(*getChildren*, *getParent*, *hasChildren*). První metodou se získávají potomci daného objektu. Zde je jasné, že Projekt má potomka Třídu a Třída má potomka Metodu. List se zde jenom převede na pole. Druhou metodou se získává rodič daného objektu a třetí metodou se kontroluje zda má objekt nějaké potomky. To stačí zkontroloval počet potomků, zda je větší než nula.

Druhá třída musí implementovat rozhraní *ITableLabelProvider*. Zde je nejdůležitější metoda *getColumnText*. Ta pro daný sloupec a pro daný objekt vrátí text, který se má zobrazit do sloupce. Zde stačí jenom zavolat na daný objekt metodu *toString*.

Pro nastavení vstupního listu se provádí metodou *setInput*.

Pro zobrazení výsledků stačí kliknout na danou metodu. To se dosáhne zaregistrování události pro dvojitý klik. Ta třídu *TreeViewer* se zavolá metoda *addDoubleClickListener*, které se předá instance třídy, která implementuje rozhraní *IDoubleClickListener*. Této třídě se pak v metodě *doubleClick* získá z parametru objekt na který se klikl. Pro zjednodušení se uzly dají

rozkliknout. To se docílí zavoláním metody *setExpandedState* na *TreeViewer*. Zde se zadá o jaký objekt(uzel) se jedná a zda se má rozbalit a nebo sbalit. Pomocí metody *getExpandedState* se získá předešlý stav.

Pokud se jedná o metodu(instance třídy *Method*, tak se vytvoří okno s výsledky analýzy a případné zobrazení výsledků(viz. kapitola 5.6.8). A jako poslední se otevře editor dané metody(třídy) pomocí metody *openEditorOfGivenProject*.

Do view se dále do „toolbar“ vloží tlačítko pro odstranění anotací(viz. kapitola 5.6.8). Pro odstranění se nejprve zjistí aktivní editor(třída *IFile*). A následně se zavolá metoda *removeAllAnnotation*, které se předá *IFile* a referenci na otevřený editor.

Tlačítko s atributem „toolbar:codecoverage.view.coverageview“. První část je, že se tlačítko má umístit jako „toolbar“ a druhá část značí, že tlačítko se má vložit do view. O tlačítko se stará třída *codecoverage.RemoveAnnotation* a má identifikátor „codecoverage.removeannotation“(Princip vkládání tlačítek viz. kapitola 5.6.2).

5.6.8 Správa anotací

Anotací se nemyslí anotace pro Javu. Anotace v prostředí Eclipse znamená zvýraznění kódu. Jako nejznámější je asi zobrazení chyby(podtržení) nebo debugování kódu(zvýraznění aktuálního řádku), vložení tzv. „breakpoint“ a podobně. Jelikož se na internetu nenachází moc návodů na vložení anotací, byla funkčnost zjištěna v modulu, který obsahuje Jacoco a projektu na stránce[7]. S anotacemi dále souvisí marker. To jsou ikony které se nacházejí vlevo od kódu(třeba ikona „breakpoint“)

Jako nejdůležitější je definování anotací(*Annotation*) a marker. To se děje v souboru „plugin.xml“. Je to o něco těžší než definování tlačítek a nebo view. Pro každou anotaci musí existovat vlastní marker.

Marker se definuje v „org.eclipse.core.resources.markers“. Jejich vlastnosti jsou:

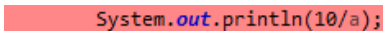
- **id** - Identifikátor marker
- **name** - Název marker
- **super type** - Z kterých typu marker bude dědit
- **persistent value** - Musí být *true*

Jelikož chceme text i obrázek, bude marker dědit z *org.eclipse.core.resources.textmarker* a *org.eclipse.core.resources.marker*. Pro účel diplomové práce se vytvoří tři marker. První(*codecoverage.markers* slouží k obyčejnému řádku(obrázek 28). Druhý(*codecoverage.markers.markerthrow*) pokud na daném řádku se vyhodila výjimka(obrázek 27) a třetí(*codecoverage.markers.markerbranch*), zda na daném řádku je větvení programu a zobrazení o celkovém počtu průchodu větví.(obrázek 29)

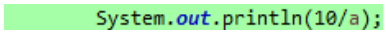
Dále se vytvoří tří anotace, ty se definují v „org.eclipse.ui.editors.markerAnnotationSpecification“. Toto značí typ anotací. Dá se vytvořit vlastní, ale pro potřeby se využije již naimplementovaná

anotace. Obsahuje dále identifikátor anotace a jméno anotace, této specifikace. Do ní se pak vypisují jednotlivé anotace. Vlastností má hodně, ale mezi nejdůležitější je barva, jakou má obarvit kód, jakým stylem obarvit(pozadí textu, podtrhnout a podobně), typ anotace(reference na *org.eclipse.ui.editors.annotationTypes*), název a ikonu. Jejich význam je stejný jako u marker.

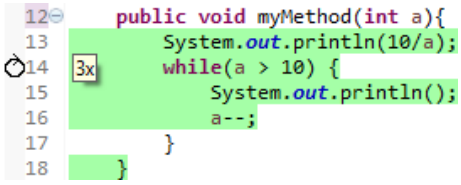
Jelikož se ikona musí zadat a v případě obyčejného příkazu se musela „nakreslit“ průhledná ikona.



Obrázek 27: Zbarvení řádku, pokud se zde vyhodila vyjímka



Obrázek 28: Zbarvení řádku, pokud příkaz proběhl v pořádku



Obrázek 29: Zobrazení o počtu průchodů v cyklu, celkově

K anotacím dále patří i typy anotací. Ty spojují anotaci s marker.

Dále je potřeba zadefinovat třídu, která se stará v případě potřeby o překreslení anotací a marker(například se přepne na editor). To se provádí v „org.eclipse.ui.editors.markerUpdaters“, kde se definuje *updater*. Ten má tři vlastnosti. První je identifikátor, třída která se využívá a pro který marker. Třída definuje metodu *updateMarker*, kde se vypočítá nová pozice marker. Začátek začátek offsetu a konec je offset plus délka pozice. Nové čísla se vloží do marker a vrátí se *true*. V metodě *getMarkerType* se vrátí typ marker.

Pro vytváření nových anotací a marker slouží třída *AnnotationFactory*. Třída obsahuje jednotlivé identifikátory anotací, marker a dále všechny anotace, které se vytvořily. Obsahuje čtyři statických metod.

- **createMarker** - Vytvoří nový marker
- **removeAllAnnotation** - Odstraní všechny anotace v dané třídě(souboru)
- **addLineAnnotation** - Vytvoření anotace pro jeden řádek(při „debugování“ pokrytí metody
- **addAnnotation** - Vytvoření anotací pro celou metodu(pokrytí řádků)

První metoda vytvoří daný typ markeru. Zde se rozlišuje, zda se jedná o řádek, kde se vyhodila výjimka(parametr *isThrowCode* nebo zda se jedná o řádkové zobrazení anotací(parametr

isLineCov). Marker se vytváří pomocí metody *createMarker* z daného souboru(třída *IResource*), které se předá identifikátor marker(z definice v souboru „plugin.xml“). A dále se mu staví parametr *IMarker.MESSAGE*, který určuje jaký text zobrazí po přejetí myší. Musí se mu nastavit i pozice v souboru. Ta se vezme z parametru(*position*) metody a vypočítaná pozice se vloží do marker.

Druhá metoda vymaže všechny anotace v souboru. Jelikož soubor může obsahovat i další anotace, musí se kontrolovat její typ a odstraníme naše vytvořené anotace. Anotace se vymažou pomocí metody *deleteMarkers*, které předáme identifikátor, zda má vymazat i v podsložkách a do jaké hloubky(v tomto případně do nekonečna). Dále musíme odstranit i anotace. Nejprve musíme získat ze souboru *IAnnotationModel*, který se stará o anotace. Připojíme se na něj pomocí metody *connect*, které předáme dokument, který chceme upravovat. Následně procházíme všechny anotace(pomocí iterátoru) a vyhledáme jenom ty, které jsem my vytvořili a ty následně odstraníme metodou *removeAnnotation*. Po skončení je důležité se odpojit metodou *disconnect*.

```
IDocumentProvider idp = editor.getDocumentProvider();
IFile file = (IFile) editor.getEditorInput().getAdapter(IFile.class);
IDocument document = idp.getDocument(editor.getEditorInput());
IAnnotationModel iamf = idp.getAnnotationModel(editor.getEditorInput());
```

Výpis 29: Získání IAnnotationModelze souboru

Třetí metoda Vytvoří anotaci pro daný řádek. Jako v předešlém případě, i zde se musíme připojit k *IAnnotationModel*. Následně si vytáhneme informace o pokrytí daného řádku. Pokud řádek je menší nulou, znamená to, že na daném řádku se vyhodila výjimka. Ze třídy *IDocument*, lze získat začátek a délku řádku v dokumentu pomocí metody *getLineInformation*. Zde je dát pozor, řádky se číslují od jedničky, ale indexování se provádí od nuly. Jelikož na daném řádku může být příkaz, který je více řádků, prochází se dané řádky a přičítá se jejich délka k celkové. Vytvoří se nový marker. Eclipse už obsahuje předpřipravené anotace, jednou z nich je *SimpleMarkerAnnotation*, které se předá identifikátor anotace(„plugin.xml“) a vytvořený marker. Vytvořená anotace se následně přidá do modelu anotací metodou *addAnnotation*, které se ještě předá pozice anotace.

Čtvrtá metoda vykreslí do editoru všechny řádky metody, které se provedly v dané cestě. Metoda je velmi podobná jako předešlá. Zde se ale vykreslují všechny řádky, které se prošly v dané cestě.

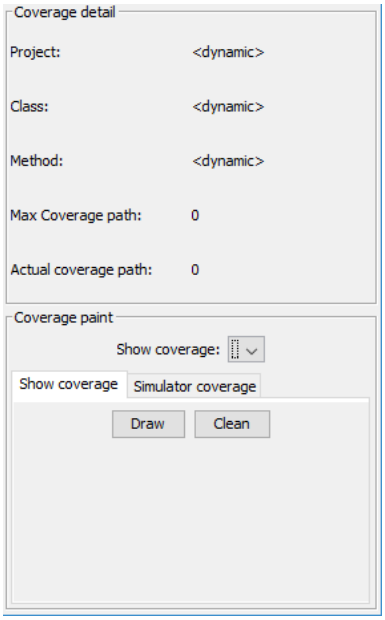
Zde se dále využívá třída *AnnotationPlugin*, která obstarává informace o modulu(například otevřené editory).

5.6.9 Zobrazení výsledků

Pro jednoduché zobrazení výsledků se využívají anotace(viz. předešlá kapitola). Po dvojitém kliku na metodu ve stromové struktury se zobrazí okno, které zobrazí informace o pokrytí cest

a následného zobrazení pokrytí. Umožňuje dvoje zobrazení. První je pro danou cestu zobrazit řádky, které se provedly a druhé, je procházejí jednotlivého řádku jak se provedl(něco jako „debugování“).

Pro jednoduchost se využívá okna typu *JFrame*. Skládá se ze čtyř částí.



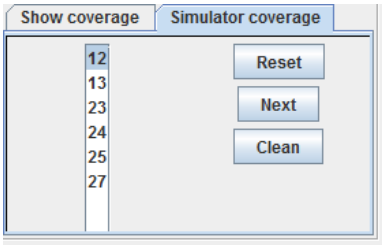
Obrázek 30: Okno pro ovládání zobrazení pokrytí a zobrazení dat o pokrytí

V první části se zobrazují informace o metodě. V jakém projektu a třídě se nachází, maximálním počtu cest(pomocí metody *getMaxPathCoverages*) a kolik cest se provedlo(pomocí metody *getActualPathCoverages*). Poslední dvě informace se nachází ve třídě *Method*.

Ve druhé části se vybírá průchod metody(všechny průchody, i ty duplicitní) pomocí „comboBox“.

Třetí část se využívá pro zobrazení pokrytí řádků pro danou cestu metodou. Zde jdou dvě tlačítka. První(„Draw“) vykreslí pokrytí, zde se využívá metoda *addAnnotation*, která vykreslí všechny řádky, které se provedl a druhé(„Clean“),které odstraní anotace v editoru(zavolá se metoda *removeAllAnnotation*).

Čtvrtá část složí k procházení dané cesty. Je zde list řádků, jak se daná cesta provedla. Dále jsou zde tři tlačítka. První tlačítko resetuje průchod a začíná se od začátku(nastaví se index aktuálního průchodu na nula). Druhé tlačítko posune index na další příkaz, který se provedl a vykreslí anotaci pomocí metody *addLineAnnotation*. Třetí tlačítko odstraní anotace z editoru(zavolá se metoda *removeAllAnnotation*).



Obrázek 31: Čtvrtá část okna pro ovládání zobrazení pokrytí

6 Závěr

Všechny požadavky na diplomovou práci byly splněny. Modul umožňuje pokrytí cest a následné vizualizace výsledků. Zde byl vybrána vizualizace výsledků přímo do kódu(editoru třídy). Nejdůležitější část diplomové práce byla získání dat o průběhu kódu. Zde bylo potřeba se naučit princip Java bytecode a JVM. Dále bylo potřeba se naučit princip vkládání bytecode pomocí knihovny ASM. Vytváření modulu pro Eclipse bylo trošku složitější, jelikož je to obrovský systém a na internetu není dostatek webů, které se této problematiky zabývají.

Jelikož se jedná o velmi složitou problematiku, modul obsahuje mnoho chyb. Většina je výsledkem špatného převodu časových značek na jednotlivé řádky. Metoda, která se k tomu využívá nebyla k tomu určena a proto některé kódy vyhodnotí špatně. Například pokud v podmínce je jenom příkaz *break* analyzátor to špatně vyhodnotí. Dále mezi chyb je potřeba v posledním bloku *case* mít alespoň jedno volání metody.

Do budoucna by se mohl naimplementovat vlastní analyzátor(převodník), který převede časové značky na jednotlivé řádky kódu. Tohle by odstranilo většinu chyb při analyzování. Dále by se mohly přidat i další metriky testování, jako například LCSAJ. V dnešní době je velmi populární *SonarQube*, proto by se mohly výsledky vyexportovat a předat je *SonarQube*. Pro časové značky se využívá metoda *System.nanoTime*. Zde může dojít teoreticky k přetečení čísla v případě velkého systému, který se analyzuje. Proto by se jako časovou značku mohla používat třídní proměnná, která se inkrementuje od nuly. Když se vloží nová značka, tak se inkrementuje. Tím by se zvýšil počet časových značek, které se dají využít v rámci třídy.

Literatura

[1] EclEmma - Java Code Coverage for Eclipse. *EclEmma - Java Code Coverage for Eclipse* [online]. Switzerland: Marc R. Hoffmann, 2009 [cit. 2017-04-03]. Dostupné z: http://www.eclemma.org/

[2] ECobertura | Eclipse Plugins, Bundles and Products - Eclipse Marketplace. *ECobertura | Eclipse Plugins, Bundles and Products - Eclipse Marketplace* [online]. Joachim Hofer, 2010 [cit. 2017-04-03]. Dostupné z: https://marketplace.eclipse.org/content/ecobertura

[3] TikiOne JaCoCoverage - NetBeans Plugin detail. TikiOne JaCoCoverage - *NetBeans Plugin detail* [online]. Jonathan Lermitage, 2013 [cit. 2017-04-03]. Dostupné z: http://plugins.netbeans.org/plugin/48570/tikione-jacocoverage

[4] JVM Internals. *JVM Internals* [online]. James D Bloom, 2013 [cit. 2017-04-09]. Dostupné z: http://blog.jamesdbloom.com/JVMInternals.html

[5] EclEmma - Java Code Coverage for Eclipse. *EclEmma - Java Code Coverage for Eclipse* [online]. Berne,Switzerland: Mountainminds GmbH & Co. KG and Contributors, 2017 [cit. 2017-04-10]. Dostupné z: http://www.jacoco.org/

[6] Java Code Example. *Java Code Example* [online]. Pivotal Software, 2012 [cit. 2017-04-25]. Dostupné z: http://www.programcreek.com/java-api-examples/index.php?source_dir=eclipse-integration-gradle-master/org.springsource.ide.eclipse.gradle.core.test/src/org/springsource/ide/eclipse/gradle/core/test/

[7] Best practices for developing Eclipse plugins. *Best practices for developing Eclipse plugins* [online]. Andy Flatt and Mickael Maison, 2011 [cit. 2017-04-26]. Dostupné z: https://www.ibm.com/developerworks/opensource/tutorials/os-eclipse-plugin-guide/

[8] Help - Eclipse Platform. *Help - Eclipse Platform* [online]. IBM corporation, 2000 [cit. 2017-04-26]. Dostupné z: http://help.eclipse.org/neon/index.jsp

[9] Your First Plug-in. *Your First Plug-in* [online]. Andrew Irvine, OTI, 2002 [cit. 2017-04-26]. Dostupné z: https://www.eclipse.org/articles/Article-Your%20First%20Plug-in/YourFirstPlugin.html

[10] PDE Does Plug-ins. *PDE Does Plug-ins* [online]. Wassim Melhem and Dejan Glozic, IBM Canada, 2003 [cit. 2017-04-26]. Dostupné z: https://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html

[11] Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model. *Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model* [online]. Lars Vogel, Simon Scholz, 2009 [cit. 2017-04-26]. Dostupné z: http://www.vogella.com/tutorials/EclipseJDT/article.html

[12] Bytecode basics | JavaWorld. *Bytecode basics | JavaWorld* [online]. Bill Venners, 1996 [cit. 2017-04-26]. Dostupné z: http://www.javaworld.com/article/2077233/core-java/bytecode-basics.html

[13] Eclipse IDE Plug-in Development: Plug-ins, Features, Update Sites and IDE Extensions. *Eclipse IDE Plug-in Development: Plug-ins, Features, Update Sites and IDE Extensions* [online]. vogella, 2009 [cit. 2017-04-26]. Dostupné z: http://www.vogella.com/tutorials/EclipsePlugin/article.html

[14] A Java Programmer's Guide to Byte Code - Beyond Java. *A Java Programmer's Guide to Byte Code - Beyond Java* [online]. Stephan Rauh, 2015 [cit. 2017-04-26]. Dostupné z: https://www.beyondjava.net/blog/java-programmers-guide-java-byte-code/

[15] Creating an Eclipse View. *Creating an Eclipse View* [online]. Dave Springgay, 2001 [cit. 2017-04-26]. Dostupné z: https://eclipse.org/articles/viewArticle/ViewArticle2.html

[16] ASM - Introduction to the ASM 2.0 Bytecode Framework. *ASM - Introduction to the ASM 2.0 Bytecode Framework* [online]. Eugene Kuleshov, 2005 [cit. 2017-04-26]. Dostupné z: http://asm.ow2.org/doc/tutorial-asm-2.0.html

[17] Java Code To Byte Code - Part One. *Java Code To Byte Code - Part One* [online]. James D Bloom, 2013 [cit. 2017-04-26]. Dostupné z: http://blog.jamesdbloom.com/JavaCodeToByteCode_PartOne.html

[18] LINDHOLM, Tim. *The Java virtual machine specification*. Java SE 8 edition. ISBN 978-0133905908.

[19] ENGEL, Joshua. *Programming for the Java virtual machine*. Reading, Mass.: Addison-Wesley, c1999. ISBN 978-0201309720.

[20] Java bytecode instruction listings. *Java bytecode instruction listings* [online]. [cit. 2017-04-26]. Dostupné z: https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

A Přílohy

- Eclipse pro spuštění modulu
- Workspace s Jacoco knihovnou a modulem
- Workspace pro spuštění modulu
- Script pro zkompilování knihovny Jacoco a zkopírování agenta